VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sneha N Shastri (1BM22CS283)

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

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Sneha N Shastri (1BM22CS283),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sunayana S Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/snehanshastri/AI

Program 1
Q. Implement Tic –Tac –Toe Game

Algorithm:
Algorithm for Tic-Tac-Toe

1. Implementation of Tic-Tac-Toe game   Date: 24|07|'24

Algorithm:

Step 1: Define a function named start_game()

Algorithm for start_game():

Step 1: Create a board by calling the create_board() function. Assign 'X' and 'O' as two players in the players list. Choose a player to start the game using the random function.

Step 2: In a while loop ask the current player to input the row and column for the next move.

Step 3: If the spot is already taken then give a message to the user else if board[row][col] = = '_' then break the loop and update the board with the player's move.

Step 4: To check if the current player won call the check_winner function. If the function returns true the player wins.

Step 5: ~~If~~ To know if the board is filled i.e it is a draw call the is_board_filled function and check if it returns true, if it returns true then it is a draw.

Step 6: ~~If~~ it's neither a win or draw switch the current_player using an if else construct.

Step 2: Define a function named create_board

Algorithm for create_board():

Step 1: In a nested for loop initialize a 3x3 grid with - to indicate empty and return the grid.

Step 3: Define a function named show_board()

Algorithm for show_board():

Step 1: Iterate through each row and join the elements ~~term~~ with a '|' using .join() and ~~return~~ print each row as a string. This is used to display the board.

Step 4: Define a function named is_board_filled (board)

Algorithm for is_board_filled (board):

Step 1: Iterate through each row using a for loop and if '-' exists return false else return true.

Step 5: Define a function named check_winner (board, player)

Algorithm for check_winner (board, player):

Step 1: if all([board[i][j] == player for j in range(3)]) or all([board[j][i] == player for j in range(3)]):
return true

To check diagonals,
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
else
return False
24/9/2024

Code:

```
import random

# Create a 2-dimensional board and initialize it with empty cells
def create_board():
    return [['-' for _ in range(3)] for _ in range(3)]

# Function to display the current board state
def show_board(board):
    for row in board:
        print(" | ".join(row))
    print()

# Function to check if the board is filled
def is_board_filled(board):
    for row in board:
        if '-' in row:
            return False
    return True

# Function to check if a player has won
def check_winner(board, player):
    # Check rows and columns
    for i in range(3):
        if all([board[i][j] == player for j in range(3)]) or all([board[j][i] == player for j in range(3)]):
            return True

    # Check diagonals
    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

# Function to start the game and play turns
def start_game():
    # Initialize board and players
    board = create_board()
    players = ['X', 'O']
    current_player = random.choice(players)

    while True:
        show_board(board)
        print(f"Player {current_player}'s turn")

        # Ask the user to select the row and column for the next move
        while True:
            try:
                row = int(input("Enter the row (0, 1, 2): "))
                col = int(input("Enter the column (0, 1, 2): "))
                if board[row][col] == '-':
                    break
                else:
                    print("This spot is already taken. Try again.")
            except (ValueError, IndexError):
                print("Invalid input. Please enter numbers between 0 and 2.")

        # Update the board with the player's move
        board[row][col] = current_player

        # Check if the current player won
        if check_winner(board, current_player):
            show_board(board)
            print(f"Player {current_player} wins!")
            break

        # Check if the board is filled
        if is_board_filled(board):
            show_board(board)
            print("It's a draw!")
            break

        # Switch to the other player
        current_player = 'O' if current_player == 'X' else 'X'

# Start the Tic-Tac-Toe game
if __name__ == "__main__":
    start_game()
```

Output:

```
- | - | -
- | - | -
- | - | -

Player X's turn
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 0
X | - | -
- | - | -
- | - | -

Player O's turn
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 1
X | - | -
- | O | -
- | - | -

Player X's turn
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 0
X | - | -
- | O | -
X | - | -

Player O's turn
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 0
X | - | -
O | O | -
X | - | -
```

```
Player X's turn
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 2
X | - | -
O | O | X
X | - | -

Player O's turn
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 0
This spot is already taken. Try again.
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 1
X | O | -
O | O | X
X | - | -

Player X's turn
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 2
X | O | X
O | O | X
X | - | -

Player O's turn
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 2
X | O | X
O | O | X
X | - | O

Player X's turn
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 1
X | O | X
O | O | X
X | X | O

It's a draw!
```

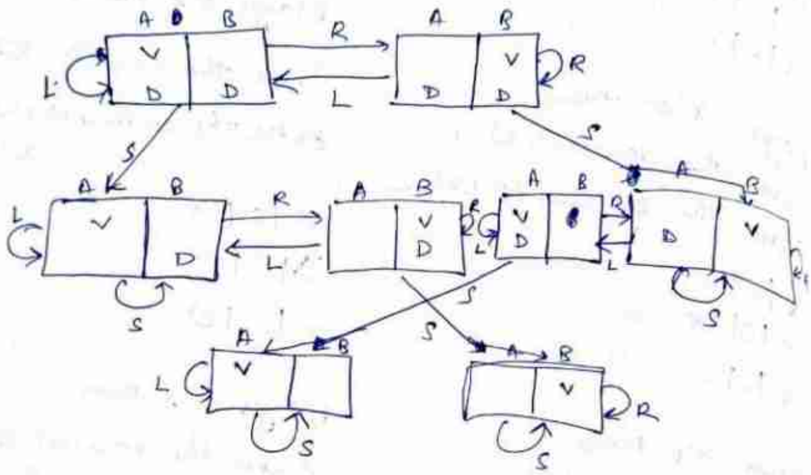
Q. Implement vacuum cleaner agent

Algorithm:

2. VACUUM WORLD CLEANER 01/10/'24

function REFLEX-VACUUM-AGENT([location, status]) returns an action
if status = Dirty then return Suck
else if location = A then return Right
else if location = B then return Left

State space diagram of Vacuum World



Parameter - Location, Status, Status of other room

Algorithm:

Step 1 - Define a function named vacuum_world()

Step 2 - Initialize the goal_state as {'A':'0', 'B':'0'} and the cost variable as 0.

Step 3 - Enter the location of the vacuum and status of current location, enter the status of other room, and store these in location_input, status_input and status_input_complement respectively.

Step 4: In an if-else construct perform the following operations:

a. If location is A and status-input = 1 it means location A is dirty and vacuum is placed there so it sucks the dust and cost is increased.

b. If location is A and status-input-complement is 1 ~~and~~ it means the other room is B and it is dirty, so the vacuum moves to B and cleans the dust so the cost is increased. But if status-input-complement is 0 then there is no action taken.

c. If location is A and status_input = 0, location A is already clean. So now status-input-complement is checked. If it is 1 then B is dirty hence vacuum moves there and cost is increased once the dust is clean. If status-input-complement = 0, B is also clean.

d. If ~~stat~~ location is not A then the vacuum cleaner is in B and the same mechanism is followed → checking of status-input for status of B and status-input_complement for status of A in this case.

Step 5: Once both A and B are clean we reach the goal state A=0 & B=0.

Code:

```
#Enter LOCATION A/B in captial letters
#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY

def vacuum_world():
      # initializing goal_state
      # 0 indicates Clean and 1 indicates Dirty
  goal_state = {'A': '0', 'B': '0'}
  cost = 0

  location_input = input("Enter Location of Vacuum") #user_input of location vacuum is placed
  status_input = input("Enter status of " + location_input) #user_input if location is dirty or clean
  status_input_complement = input("Enter status of other room")
  other_location=''
  if location_input=='A':
   other_location='B'
  else:
   other_location='A'

  initial_state = {location_input: status_input, other_location: status_input_complement}
  print("Initial Location Condition" + str(initial_state))

  if location_input == 'A':
    # Location A is Dirty.
    print("Vacuum is placed in Location A")
    if status_input == '1':
      print("Location A is Dirty.")
      # suck the dirt  and mark it as clean
      goal_state['A'] = '0'
      cost += 1                    #cost for suck
      print("Cost for CLEANING A " + str(cost))
      print("Location A has been Cleaned.")

      if status_input_complement == '1':
        # if B is Dirty
        print("Location B is Dirty.")
        print("Moving right to the Location B. ")
        #cost += 1                    #cost for moving right
        print("COST for moving RIGHT" + str(cost))
        # suck the dirt and mark it as clean
        goal_state['B'] = '0'
        cost += 1                    #cost for suck
        print("COST for SUCK " + str(cost))
        print("Location B has been Cleaned. ")
      else:
        print("No action" + str(cost))
        # suck and mark clean
```

```
            print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean ")
            if status_input_complement == '1':# if B is Dirty
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                #cost += 1                  #cost for moving right
                print("COST for moving RIGHT " + str(cost))
                # suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1                   #cost for suck
                print("Cost for SUCK" + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action " + str(cost))
                print(cost)
                # suck and mark clean
                print("Location B is already clean.")

    else:
        print("Vacuum is placed in location B")
        # Location B is Dirty.
        if status_input == '1':
            print("Location B is Dirty.")
            # suck the dirt  and mark it as clean
            goal_state['B'] = '0'
            cost += 1  # cost for suck
            print("COST for CLEANING " + str(cost))
            print("Location B has been Cleaned.")

            if status_input_complement == '1':
                # if A is Dirty
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                #cost += 1  # cost for moving right
                print("COST for moving LEFT" + str(cost))
                # suck the dirt and mark it as clean
                goal_state['A'] = '0'
                cost += 1  # cost for suck
                print("COST for SUCK " + str(cost))
                print("Location A has been Cleaned.")

        else:
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")
```

```
        if status_input_complement == '1':  # if A is Dirty
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            #cost += 1  # cost for moving right
            print("COST for moving LEFT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
        else:
            print("No action " + str(cost))
            # suck and mark clean
            print("Location A is already clean.")

    # done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))


vacuum_world()
```

Output:

```
Enter Location of VacuumA
Enter status of A0
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '1'}
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 0
Cost for SUCK1
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
```

Program 2:

Q. Implement 8-Puzzle using BFS, DFS.

Algorithm:

Algorithm for BFS:

8/11/2024

Q. Solve 8 puzzle problem using BFS and DFS

i) Using BFS

~~Step 1: Define a class Node which represents a state~~

i.e.

```
        1 2 3
        4 5 6
        7 8 0
   up /       \ left
 1 2 3         1 2 3
 4 5 0         4 5 6
 7 8 6         7 0 8
               Goal State
```

Algorithm:

Step 1: ~~Represent~~ Define a class Node which represents a state in the search space

Step 2: Define a class called Queue Frontier which contains the queue operations.

i. add() - Adds a node to the frontier

ii. contains - state() - Checks if the frontier contains a given state

iii. empty() - Returns true if the frontier is empty

iv. remove() - Removes and returns the first node from the frontier

Step 3: Define a class Puzzle

i. init() - Initializes the start state, goal state and solution.

ii. neighbours () - returns valid moves from a given state.

iii. does_not_contain_state (): checks if a state has already been explored.

iv. solve ( ) - This is the function that performs BFS algorithm to find a solution.
→ It initializes the frontier with start state and list of explored states as empty.
→ It iteratively explores states from the frontier checking if they match the goal state.
→ If a solution is found it reconstructs the path and restores it.
→ If the frontier becomes empty without finding a solution, it indicates that no solution exists.

Output:
Enter initial state
1 2 3
4 5 6
7 8 0
Enter goal state
1 2 3
4 5 6
7 0 8

Algorithm for DFS:

Algorithm:

Step 1: Define a class Node which represents the state in the search space.

Step 2: Define a class called Stack Frontier which contains the stack operations:

~~i. add() - Adds~~

Step 3: Add the initial node to the frontier. Create an empty set explored to store the explored states.

Step 4: While the frontier is not empty:

→ Remove the last node added to the frontier

→ Increment the number of explored states

→ If the current node's state matches the goal state:

- Reconstruct the solution path by backtracking from the current node to initial node.
- Return solution path.

→ If the current node's state has not been explored:

- Add the current node's state to the explored ~~states~~ set.
- Generate the neighbours of the current state using neighbours() method.
- For each neighbour:

→ If the neighbour's state is not in the frontier and has not been explored:

Create a new Node object for the neighbour

Add the neighbour node to the frontier

→ No Solution

- If the frontier becomes empty and goal state has not been reached, raise an exception indicating that no solution was found.

Code:

BFS:

```
import numpy as np #bfs
from collections import deque


class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action


class QueueFrontier:
    def __init__(self):
        self.frontier = deque()  # Use deque for efficient queue operations

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            return self.frontier.popleft()  # Remove from the front of the queue


class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None

    def neighbors(self, state):
        mat, (row, col) = state
```

```
    results = []

    if row > 0:  # Move up
      mat1 = np.copy(mat)
      mat1[row][col] = mat1[row - 1][col]
      mat1[row - 1][col] = 0
      results.append(('up', [mat1, (row - 1, col)]))
    if col > 0:  # Move left
      mat1 = np.copy(mat)
      mat1[row][col] = mat1[row][col - 1]
      mat1[row][col - 1] = 0
      results.append(('left', [mat1, (row, col - 1)]))
    if row < 2:  # Move down
      mat1 = np.copy(mat)
      mat1[row][col] = mat1[row + 1][col]
      mat1[row + 1][col] = 0
      results.append(('down', [mat1, (row + 1, col)]))
    if col < 2:  # Move right
      mat1 = np.copy(mat)
      mat1[row][col] = mat1[row][col + 1]
      mat1[row][col + 1] = 0
      results.append(('right', [mat1, (row, col + 1)]))

    return results

  def print_solution(self):
    if self.solution is not None:
      print("Start State:\n", self.start[0], "\n")
      print("Goal State:\n", self.goal[0], "\n")
      print("\nStates Explored: ", len(self.explored), "\n")
      print("Solution:\n ")
      for action, cell in zip(self.solution[0], self.solution[1]):
        print("Action: ", action, "\n", cell[0], "\n")
      print("Goal Reached!!")
    else:
      print("No solution found.")

  def does_not_contain_state(self, state):
    for st in self.explored:
      if (st[0] == state[0]).all():
        return False
    return True
```

```
def solve(self):
    self.num_explored = 0

    start = Node(state=self.start, parent=None, action=None)
    frontier = QueueFrontier()  # Use QueueFrontier for BFS
    frontier.add(start)

    self.explored = []

    while True:
        if frontier.empty():
            raise Exception("No solution")

        node = frontier.remove()
        self.num_explored += 1

        # Display the explored state and the action taken to get there
        if node.parent is not None:
            print(f"Exploring state after moving '{node.action}':\n{node.state[0]}\n")

        if (node.state[0] == self.goal[0]).all():
            actions = []
            cells = []
            while node.parent is not None:
                actions.append(node.action)
                cells.append(node.state)
                node = node.parent
            actions.reverse()
            cells.reverse()
            self.solution = (actions, cells)
            return

        # Add the current state to explored states only once
        if not any((ex_state[0] == node.state[0]).all() for ex_state in self.explored):
            self.explored.append(node.state)

        for action, state in self.neighbors(node.state):
            # Check if state is the initial state
            if not np.array_equal(state[0], self.start[0]) and \
                    not frontier.contains_state(state) and self.does_not_contain_state(state):
                child = Node(state=state, parent=node, action=action)
                frontier.add(child)
```

```
def input_puzzle():
    print("Enter the initial state (3x3 matrix, use spaces to separate numbers):")
    start = np.array([list(map(int, input().split())) for _ in range(3)])

    print("Enter the goal state (3x3 matrix, use spaces to separate numbers):")
    goal = np.array([list(map(int, input().split())) for _ in range(3)])

    startIndex = tuple(map(int, np.argwhere(start == 0)[0]))  # Find position of 0 in start
    goalIndex = tuple(map(int, np.argwhere(goal == 0)[0]))  # Find position of 0 in goal

    return start, startIndex, goal, goalIndex


if __name__ == "__main__":
    start, startIndex, goal, goalIndex = input_puzzle()
    p = Puzzle(start, startIndex, goal, goalIndex)
    p.solve()
    p.print_solution()
```

Output (BFS):

```
Enter the initial state (3x3 matrix, use spaces to separate numbers):
1 2 3
4 5 6
7 8 0
Enter the goal state (3x3 matrix, use spaces to separate numbers):
1 2 3
4 5 6
7 0 8
Exploring state after moving 'up':
[[1 2 3]
 [4 5 0]
 [7 8 6]]

Exploring state after moving 'left':
[[1 2 3]
 [4 5 6]
 [7 0 8]]

Start State:
 [[1 2 3]
 [4 5 6]
 [7 8 0]]
```

Goal State:
[[1 2 3]
[4 5 6]
[7 0 8]]

States Explored: 2

Solution:

Action: left
[[1 2 3]
[4 5 6]
[7 0 8]]

Goal Reached!!

DFS:

```
import numpy as np #dfs


class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action


class StackFrontier:
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)

    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0

    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
```

```
        node = self.frontier[-1]
        self.frontier = self.frontier[:-1]
        return node


class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None

    def neighbors(self, state):
        mat, (row, col) = state
        results = []

        if row > 0:  # Move up
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row - 1][col]
            mat1[row - 1][col] = 0
            results.append(('up', [mat1, (row - 1, col)]))
        if col > 0:  # Move left
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col - 1]
            mat1[row][col - 1] = 0
            results.append(('left', [mat1, (row, col - 1)]))
        if row < 2:  # Move down
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row + 1][col]
            mat1[row + 1][col] = 0
            results.append(('down', [mat1, (row + 1, col)]))
        if col < 2:  # Move right
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col + 1]
            mat1[row][col + 1] = 0
            results.append(('right', [mat1, (row, col + 1)]))

        return results

    def print_solution(self):
        if self.solution is not None:
            print("Start State:\n", self.start[0], "\n")
            print("Goal State:\n", self.goal[0], "\n")
            print("\nStates Explored: ", self.num_explored, "\n")
            print("Solution:\n ")
            for action, cell in zip(self.solution[0], self.solution[1]):
                print("Action: ", action, "\n", cell[0], "\n")
            print("Goal Reached!!")
```

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

    def solve(self):
        self.num_explored = 0
        start = Node(state=self.start, parent=None, action=None)
        frontier = StackFrontier()  # Use StackFrontier for DFS
        frontier.add(start)

        self.explored = set()  # Use a set to track explored states

        while True:
            if frontier.empty():
                raise Exception("No solution")

            node = frontier.remove()
            self.num_explored += 1

            # Display the explored state and the action taken to get there
            if node.parent is not None:
                print(f"Exploring state after moving '{node.action}':\n{node.state[0]}\n")

            if (node.state[0] == self.goal[0]).all():
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                return

            # Convert state to a tuple for set operations
            state_tuple = tuple(map(tuple, node.state[0]))
            if state_tuple not in self.explored:
                self.explored.add(state_tuple)  # Add to explored set

                for action, state in self.neighbors(node.state):
                    child_state_tuple = tuple(map(tuple, state[0]))
                    if not frontier.contains_state(state) and child_state_tuple not in self.explored:
                        child = Node(state=state, parent=node, action=action)
                        frontier.add(child)


def input_puzzle():
```

```python
    print("Enter the initial state (3x3 matrix, use spaces to separate numbers):")
    start = np.array([list(map(int, input().split())) for _ in range(3)])

    print("Enter the goal state (3x3 matrix, use spaces to separate numbers):")
    goal = np.array([list(map(int, input().split())) for _ in range(3)])

    startIndex = tuple(map(int, np.argwhere(start == 0)[0]))  # Find position of 0 in start
    goalIndex = tuple(map(int, np.argwhere(goal == 0)[0]))  # Find position of 0 in goal

    return start, startIndex, goal, goalIndex


if __name__ == "__main__":
    start, startIndex, goal, goalIndex = input_puzzle()
    p = Puzzle(start, startIndex, goal, goalIndex)
    p.solve()
    p.print_solution()
```

Output (DFS):

```
Enter the initial state (3x3 matrix, use spaces to separate numbers):
1 2 3
4 5 6
7 0 8
Enter the goal state (3x3 matrix, use spaces to separate numbers):
1 2 3
4 5 6
7 8 0
Exploring state after moving 'right':
[[1 2 3]
 [4 5 6]
 [7 8 0]]

Start State:
 [[1 2 3]
 [4 5 6]
 [7 0 8]]

Goal State:
 [[1 2 3]
 [4 5 6]
 [7 8 0]]


States Explored:  2
```

Solution:

Action: right
 [[1 2 3]
 [4 5 6]
 [7 8 0]]

Goal Reached!!

Program 3:

Q. Implement A* Search Algorithm - Misplaced Tiles and Manhattan

Algorithm:

Algorithm for A* Search

Algorithm of A* search:

Step 1: Place the starting node in the OPEN list.

Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stop.

Step 3: Select the node from the OPEN list which has the smallest value of evaluation function. (g+h) if node n is goal node then, return success and stop, otherwise

Step 4: Expand node n and generate all of its successors and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into OPEN list.

Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value

Step 6: Return to Step 2.

Algorithm for A* Misplaced Tiles:

Algorithm for Heuristic approach – ~~Missing~~ Misplaced tiles method

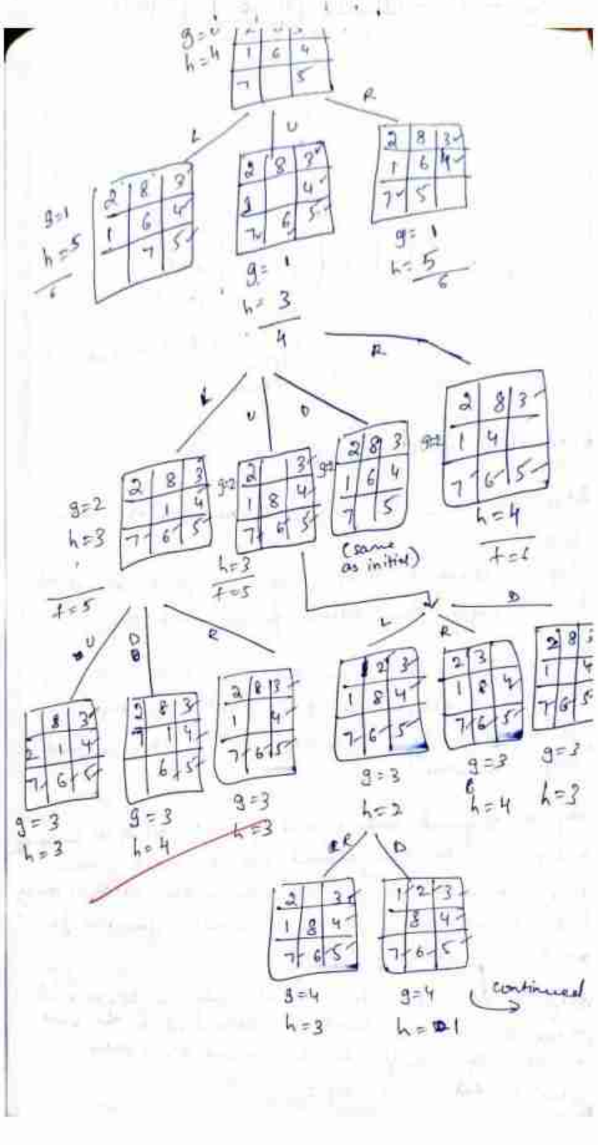
Step 1: Define g for each level which is nothing but the depth. Define h for each level, where h represents the no. of misplaced tiles

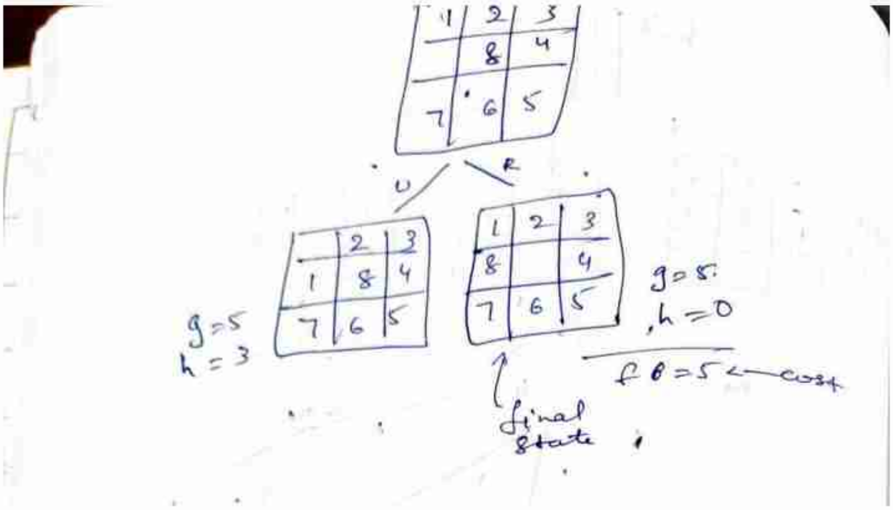
Step 2: Calculate the cost function i.e $f(n) = g(n) + h(n)$.

Step 3: Now the minimum value of $f(n)$ is explored.

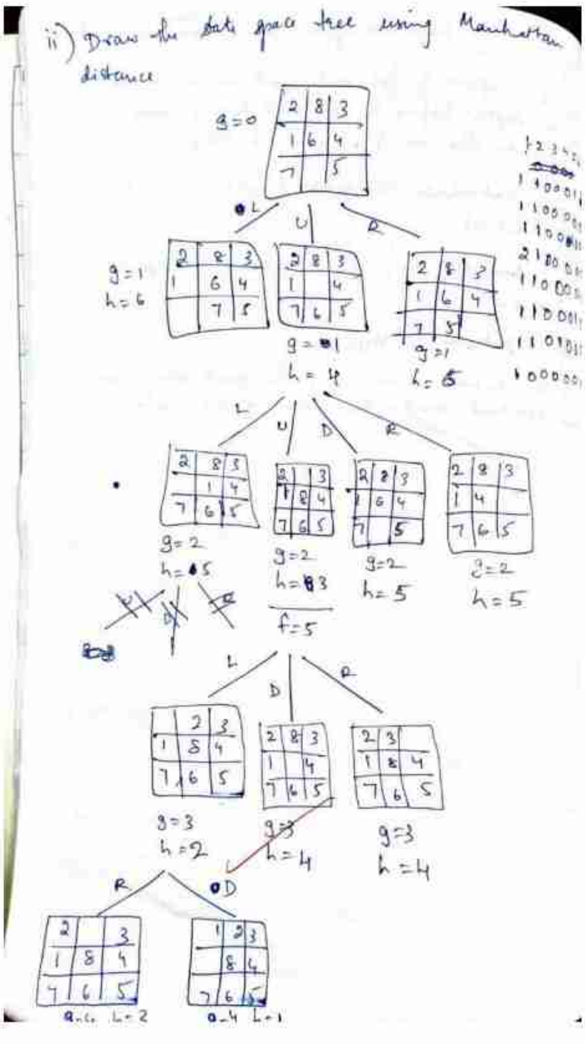
Step 4: Return to step 2.

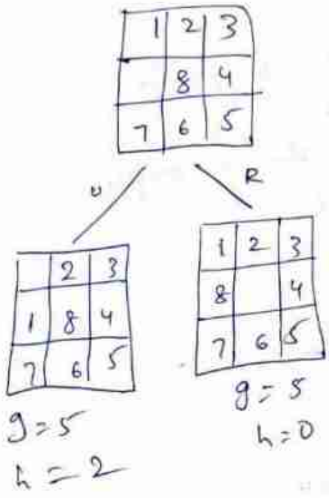
Step 5: When $h(n) = 0$, the goal state has been reached and the cost $= g(n) + h(n)$.





Algorithm for A* Manhattan Distance:





Algorithm for Heuristic approach: Manhattan distance

Step 1: Define g for each level which is nothing but the depth. Define h for each state which is the Manhattan distance.

Step 2: The Manhattan distance is the sum of the difference in the no. of moves required for a number to move from its initial position to reach the goal position. This manhattan distance is stored in h(n).

Step 3: Calculate the cost, f(n) = g(n) + h(n). The state with minimum cost is explored further.

Step 4: Return to step 2

Step 5: When h(n) = 0, the goal state is reached and the cost = g(n) + h(n).

Functions:

manhattan_distance() - To find the manhattan distance.

Code:

A* Misplaced Tiles

```
import heapq
```

```
# Function to check if the puzzle is in the goal state
def is_goal(state, goal_state):
    return state == goal_state

# Function to calculate the Misplaced Tiles heuristic
def misplaced_tiles(state, goal_state):
    count = 0
    for i in range(len(state)):
        if state[i] != goal_state[i] and state[i] != 0:  # Exclude the blank tile (0)
            count += 1
    return count

# Function to find possible moves (successors) from the current state
def get_successors(state):
    successors = []
    blank_idx = state.index(0)  # Find the index of the blank (0)

    # Possible moves: up, down, left, right
    moves = {
        "up": -3, "down": 3, "left": -1, "right": 1
    }

    for direction, move in moves.items():
        new_idx = blank_idx + move
        if 0 <= new_idx < 9:  # Check if the move is within bounds
            if direction == "left" and blank_idx % 3 == 0:
                continue  # Skip invalid move to the left
            if direction == "right" and blank_idx % 3 == 2:
                continue  # Skip invalid move to the right
            new_state = state[:]
            new_state[blank_idx], new_state[new_idx] = new_state[new_idx], new_state[blank_idx]
            successors.append(new_state)

    return successors

# A* Algorithm using Misplaced Tiles heuristic
def astar_misplaced_tiles(start_state, goal_state):
    open_list = []
    closed_list = set()

    # Push the initial state into the priority queue (heap), with f = g + h
    initial_h = misplaced_tiles(start_state, goal_state)
```

```
    heapq.heappush(open_list, (initial_h, 0, start_state, []))

    print(f"\nLevel 0 (Initial State):")
    display_states_side_by_side([start_state])  # Display the start state
    print(f"g(n) = 0, h(n) = {initial_h}, f(n) = {initial_h}\n")

    level = 1  # Track levels for display

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

        if is_goal(current_state, goal_state):
            return path + [current_state]  # Return the path when goal is reached

        if tuple(current_state) in closed_list:
            continue

        closed_list.add(tuple(current_state))

        # Expand the current node (find successors)
        level_states = []
        for successor in get_successors(current_state):
            if tuple(successor) not in closed_list:
                new_g = g + 1  # Increment the cost to reach the successor
                new_h = misplaced_tiles(successor, goal_state)
                new_f = new_g + new_h
                heapq.heappush(open_list, (new_f, new_g, successor, path + [current_state]))
                level_states.append((successor, new_g, new_h, new_f))

        if level_states:
            print(f"\nLevel {level}:")
            for state_info in level_states:
                state, new_g, new_h, new_f = state_info
                display_states_side_by_side([state])
                print(f"g(n) = {new_g}, h(n) = {new_h}, f(n) = {new_f}\n")
            level += 1

    return None  # No solution found

# Function to display the 8-puzzle in a readable format, multiple states side by side
def display_states_side_by_side(states):
    lines = [""] * 3  # Each puzzle has 3 lines
```

```
    for state in states:
        for i in range(0, 9, 3):
            lines[i // 3] += f"{state[i:i+3]}    "

    for line in lines:
        print(line)

# Main function to take input and run the A* algorithm
def main():
    print("Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    start_state = list(map(int, input().split()))

    print("Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    goal_state = list(map(int, input().split()))

    print("\nSolving the 8-puzzle...\n")

    solution = astar_misplaced_tiles(start_state, goal_state)

    if solution:
        print("\nSolution Path Found!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            display_states_side_by_side([state])
    else:
        print("No solution found.")

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

Output :

```
Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):
2 8 3 1 6 4 7 0 5
Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):
1 2 3 8 0 4 7 6 5

Solving the 8-puzzle...
```

Level 0 (Initial State):
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
g(n) = 0, h(n) = 4, f(n) = 4

Level 1:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
g(n) = 1, h(n) = 3, f(n) = 4

[2, 8, 3]
[1, 6, 4]
[0, 7, 5]
g(n) = 1, h(n) = 5, f(n) = 6

[2, 8, 3]
[1, 6, 4]
[7, 5, 0]
g(n) = 1, h(n) = 5, f(n) = 6

Level 2:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 2, h(n) = 3, f(n) = 5

[2, 8, 3]
[0, 1, 4]
[7, 6, 5]
g(n) = 2, h(n) = 3, f(n) = 5

[2, 8, 3]
[1, 4, 0]
[7, 6, 5]
g(n) = 2, h(n) = 4, f(n) = 6

Level 3:

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 2, f(n) = 5

[2, 3, 0]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 4, f(n) = 7

Level 4:
[0, 8, 3]
[2, 1, 4]
[7, 6, 5]
g(n) = 3, h(n) = 3, f(n) = 6

[2, 8, 3]
[7, 1, 4]
[0, 6, 5]
g(n) = 3, h(n) = 4, f(n) = 7

Level 5:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
g(n) = 4, h(n) = 1, f(n) = 5

Level 6:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]
g(n) = 5, h(n) = 2, f(n) = 7

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
g(n) = 5, h(n) = 0, f(n) = 5

Solution Path Found!

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

Code:
A* Manhattan Distance

```
import heapq

# Function to check if the puzzle is in the goal state
def is_goal(state, goal_state):
    return state == goal_state

# Function to calculate the Manhattan distance heuristic
def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(1, 9):  # Skip the blank tile (0)
        current_index = state.index(i)
        goal_index = goal_state.index(i)
        current_row, current_col = divmod(current_index, 3)
        goal_row, goal_col = divmod(goal_index, 3)
        distance += abs(current_row - goal_row) + abs(current_col - goal_col)
```

```
    return distance

# Function to find possible moves (successors) from the current state
def get_successors(state):
    successors = []
    blank_idx = state.index(0)  # Find the index of the blank (0)

    # Possible moves: up, down, left, right
    moves = {
        "up": -3, "down": 3, "left": -1, "right": 1
    }

    for direction, move in moves.items():
        new_idx = blank_idx + move
        if 0 <= new_idx < 9:  # Check if the move is within bounds
            if direction == "left" and blank_idx % 3 == 0:
                continue  # Skip invalid move to the left
            if direction == "right" and blank_idx % 3 == 2:
                continue  # Skip invalid move to the right
            new_state = state[:]
            new_state[blank_idx], new_state[new_idx] = new_state[new_idx], new_state[blank_idx]
            successors.append(new_state)

    return successors

# A* Algorithm using Manhattan Distance heuristic
def astar_manhattan_distance(start_state, goal_state):
    open_list = []
    closed_list = set()

    # Push the initial state into the priority queue (heap), with f = g + h
    initial_h = manhattan_distance(start_state, goal_state)
    heapq.heappush(open_list, (initial_h, 0, start_state, []))

    print(f"\nLevel 0 (Initial State):")
    display_states_side_by_side([start_state])  # Display the start state
    print(f"g(n) = 0, h(n) = {initial_h}, f(n) = {initial_h}\n")

    level = 1  # Track levels for display

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

```
        if is_goal(current_state, goal_state):
            return path + [current_state]  # Return the path when goal is reached

        if tuple(current_state) in closed_list:
            continue

        closed_list.add(tuple(current_state))

        # Expand the current node (find successors)
        level_states = []
        for successor in get_successors(current_state):
            if tuple(successor) not in closed_list:
                new_g = g + 1  # Increment the cost to reach the successor
                new_h = manhattan_distance(successor, goal_state)
                new_f = new_g + new_h
                heapq.heappush(open_list, (new_f, new_g, successor, path + [current_state]))
                level_states.append((successor, new_g, new_h, new_f))

        if level_states:
            print(f"\nLevel {level}:")
            for state_info in level_states:
                state, new_g, new_h, new_f = state_info
                display_states_side_by_side([state])
                print(f"g(n) = {new_g}, h(n) = {new_h}, f(n) = {new_f}\n")
            level += 1

    return None  # No solution found

# Function to display the 8-puzzle in a readable format, multiple states side by side
def display_states_side_by_side(states):
    lines = [""] * 3  # Each puzzle has 3 lines

    for state in states:
        for i in range(0, 9, 3):
            lines[i // 3] += f"{state[i:i+3]}    "

    for line in lines:
        print(line)

# Main function to take input and run the A* algorithm
def main():
    print("Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    start_state = list(map(int, input().split()))
```

```
    print("Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):")
    goal_state = list(map(int, input().split()))

    print("\nSolving the 8-puzzle...\n")

    solution = astar_manhattan_distance(start_state, goal_state)

    if solution:
        print("\nSolution Path Found!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            display_states_side_by_side([state])
    else:
        print("No solution found.")

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

Output:

```
Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):
2 8 3 1 6 4 7 0 5
Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):
1 2 3 8 0 4 7 6 5

Solving the 8-puzzle...


Level 0 (Initial State):
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
g(n) = 0, h(n) = 5, f(n) = 5


Level 1:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
g(n) = 1, h(n) = 4, f(n) = 5

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

g(n) = 1, h(n) = 6, f(n) = 7

[2, 8, 3]
[1, 6, 4]
[7, 5, 0]
g(n) = 1, h(n) = 6, f(n) = 7

Level 2:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 2, h(n) = 3, f(n) = 5

[2, 8, 3]
[0, 1, 4]
[7, 6, 5]
g(n) = 2, h(n) = 5, f(n) = 7

[2, 8, 3]
[1, 4, 0]
[7, 6, 5]
g(n) = 2, h(n) = 5, f(n) = 7

Level 3:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 2, f(n) = 5

[2, 3, 0]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 4, f(n) = 7

Level 4:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
g(n) = 4, h(n) = 1, f(n) = 5

Level 5:
[1, 2, 3]
[7, 8, 4]

[0, 6, 5]
g(n) = 5, h(n) = 2, f(n) = 7

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
g(n) = 5, h(n) = 0, f(n) = 5

Solution Path Found!
Step 0:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Step 1:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Step 2:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Step 3:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Step 4:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
Step 5:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

Program 4:

Q.Implement Iterative Deepening for 8 Puzzle and Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Algorithm for IDDFS:

22/10/2024

Q. Implement Iterative Deepening Algorithm

Algorithm: Iterative Deepening DFS

function ITERATIVE-DEEPENING-SEARCH (problem) returns a solution, or failure

for depth = 0 to ∞ do

  result ← DEPTH-LIMITED-SEARCH (problem, depth)

  if result ≠ cutoff then return result

1. For each child of the current node,
   - → If it is the largest node return
   - → If the current max depth is reached return
   - → Set the current node to this node and go back to 1.
   - → After having gone through all children go to the next child of the parent. (the next sibling)
   - → After having gone through all children of the start nodes, increase the maximum depth and go back to 1.
   - → If we have reached all leaf (bottom) nodes, the goal node doesn't exist.

State Space: (Level = 1)

```
          1 2 3
          4 5 6
          7 0 8
       ↙    ↓    ↘
1 2 3     1 2 3     1 2 3
4 5 6     4 0 6     4 5 6
0 7 8     7 5 8     7 8 0  (Goal State reached)
```

IDDFS Algorithm

Step 1: Input ~~initial~~ start_state and goal-state. Set max-depth to a limit.

step 2: For depth from 0 to max_depth,
call depth-limited-search () with initial state, goal state and current depth. If a solution is found in depth-limited-search, return the solution.

step 3: Define a depth-limited-search recursive function
If the current state is goal state and depth is 0 return current state.

If depth is >0 :
For each successor state :
Recursively call depth-limited-search () with the successor, goal state and depth -1.
If a solution is found in the recursive call return the solution.
If no solution return None.

Algorithm for Hill Climbing:

22/10/2024

Q Implement Hill Climbing Search Algorithm to solve N-Queens problem

Algorithm:

function HILL-CLIMBING(problem) returns a state that is a local maximum

current ← MAKE-NODE(problem.INITIAL-STATE)

loop do

neighbor ← a highest-valued successor of current

if neighbor.VALUE ≤ current.VALUE then return current.STATE

current ← neighbor

State: 4 queens on the board. One queen per column

- Variables: x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assume that there is one queen per column.
- Domain for each variable: $x_i \in \{0,1,2,3\}, \forall i$

Initial state: A random state

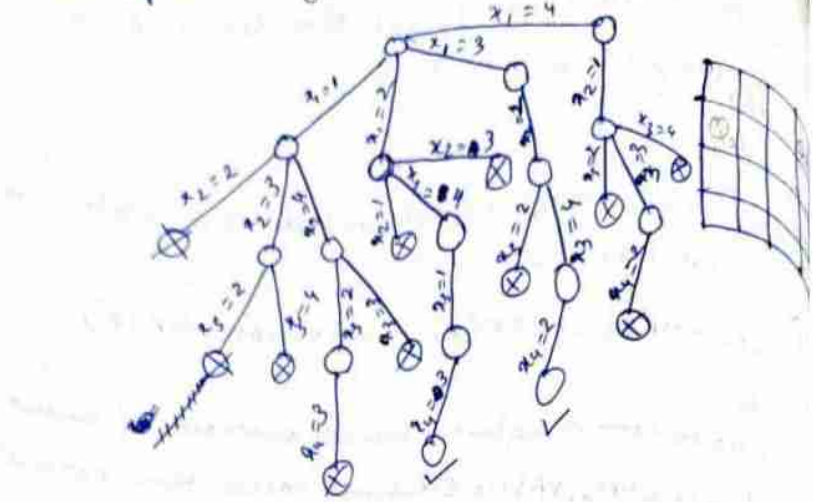
Goal state: 4 queens on the board. No pair of queens are attacking each other.

Neighbour relation:

Swap the row position of two queens.

Cost Function: The no. of pairs of queens attacking each other, directly or indirectly.

State-space tree for 4 Queens



Solutions: 2, 4, 1, 3
3, 1, 4, 2

	Q1		
			Q2
Q3			
		Q4	

		Q1	
Q2			
			Q3
	Q4		

Specific Algorithm: Hill Climbing

1. Start with empty N × N board
2. Place a queen in the first row
3. Recursively try to place the queen in the next row in a safe position
4. If safe position is found, move to next row.
5. If no safe position is found, backtrack to previous row and try different column.
6. Repeat steps 3-5 until all rows are filled or no solution.
7. If all rows filled, solution found
8. Print solution.

Code:

IDDFS

```
from copy import deepcopy

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

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

    def get_blank_position(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()

        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))

        return successors

    def is_goal(self, goal_state):
        return self.board == goal_state

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
```

```
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
    return None

def iterative_deepening_search(start_state, goal_state, max_depth):
    for depth in range(max_depth + 1):
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
    return None

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)

    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    max_depth = int(input("Enter the maximum depth for search: "))

    return start_state, goal_state, max_depth

def main():
    start_board, goal_board, max_depth = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    result = iterative_deepening_search(start_state, goal_state, max_depth)

    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
```

```
        print("Goal state not found within the specified depth.")

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

Output:

Enter the start state of the 8-puzzle (9 numbers, use 0 for the blank tile):
2 8 3 1 6 4 7 0 5
Enter the goal state of the 8-puzzle (9 numbers, use 0 for the blank tile):
1 2 3 8 0 4 7 6 5

Solving the 8-puzzle...

Level 0 (Initial State):
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
g(n) = 0, h(n) = 5, f(n) = 5

Level 1:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
g(n) = 1, h(n) = 4, f(n) = 5

[2, 8, 3]
[1, 6, 4]
[0, 7, 5]
g(n) = 1, h(n) = 6, f(n) = 7

[2, 8, 3]
[1, 6, 4]
[7, 5, 0]
g(n) = 1, h(n) = 6, f(n) = 7

Level 2:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 2, h(n) = 3, f(n) = 5

[2, 8, 3]
[0, 1, 4]

[7, 6, 5]
g(n) = 2, h(n) = 5, f(n) = 7

[2, 8, 3]
[1, 4, 0]
[7, 6, 5]
g(n) = 2, h(n) = 5, f(n) = 7

Level 3:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 2, f(n) = 5

[2, 3, 0]
[1, 8, 4]
[7, 6, 5]
g(n) = 3, h(n) = 4, f(n) = 7

Level 4:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
g(n) = 4, h(n) = 1, f(n) = 5

Level 5:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]
g(n) = 5, h(n) = 2, f(n) = 7

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
g(n) = 5, h(n) = 0, f(n) = 5

Solution Path Found!
Step 0:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Step 1:
[2, 8, 3]

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

Hill Climbing - N Queens

Code:

```
N = 4

def print_board(solution):
    for row in solution:
        print(" ".join("Q" if col else "." for col in row))
    print()

def is_safe(board, row, col):
    # Check this column on upper side
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper diagonal on left side
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if j < 0:
            break
        if board[i][j] == 1:
            return False

    # Check upper diagonal on right side
    for i, j in zip(range(row, -1, -1), range(col, N)):
```

```
        if j >= N:
            break
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens(board, row, solutions):
    if row == N:
        solutions.append([row[:] for row in board])
        return

    for col in range(N):
        if is_safe(board, row, col):
            board[row][col] = 1  # Place queen
            solve_n_queens(board, row + 1, solutions)  # Recur to place rest
            board[row][col] = 0  # Backtrack

def main():
    board = [[0 for _ in range(N)] for _ in range(N)]
    solutions = []

    solve_n_queens(board, 0, solutions)

    print(f"Found {len(solutions)} solutions:")
    for solution in solutions:
        print_board(solution)

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

Output:

```
Found 2 solutions:
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .
```

Program 5:

Q. Simulated Annealing to Solve 8-Queens problem

Algorithm:

29/10/2024

Q. write a program to implement Simulated Annealing Algorithm

Algorithm:

function SIMULATED-ANNEALING(problem, schedule)

returns a solution state

input: problem, a problem

schedule, i.e mapping from time to "temperature"

current <— MAKE – NODE(problem. INITIAL – STATE)

for t=1 to ∞ do

T <— schedule (t)

if T=0 then return current

next <— a randomly selected successor of current

ΔE <— next.VALUE – current. VALUE

if ΔE>0 then current <— next

else current <— next only with probability $e^{\Delta E/T}$

Simulated Annealing detailed algorithm:

1. Start at a random point x.
2. Choose a new point x_j on a neighbourhood N(x).
3. Decide whether or not to move to the new point x_j. The decision will be made based on the probability function $P(x, x_j, T)$

$$P(x, x_j, T) = \begin{cases} 1 & F(x_j) \geq F(x) \\ e^{\frac{F(x_j) - F(x)}{T}} & F(x_j) < F(x) \end{cases}$$

4. Reduce T

Pseudo code Algorithm:

```
FUNCTION simulated_annealing(n):
    current_solution = create_initial_solution(n)
    current_fitness = calculate_fitness(current_solution)
    best_solution = current_solution
    best_fitness = current_fitness
    temperature = initial_temp

WHILE temperature > final_temp:
    neighbor = random_neighbor(current_solution)
    neighbor_fitness = calculate_fitness(neighbor)

    IF neighbor_fitness < current_fitness OR random_
    uniform(0, 1) < exp((current_fitness - neighbor_
    fitness) / temperature):
        current_solution = neighbor
        current_fitness = neighbor_fitness

        IF current_fitness < best_fitness:
            best_solution = current_solution
            best_fitness = current_fitness

    temperature = temperature * cooling_rate
RETURN best_solution, best_fitness

PRINT best_solution, best_fitness
```

Code:

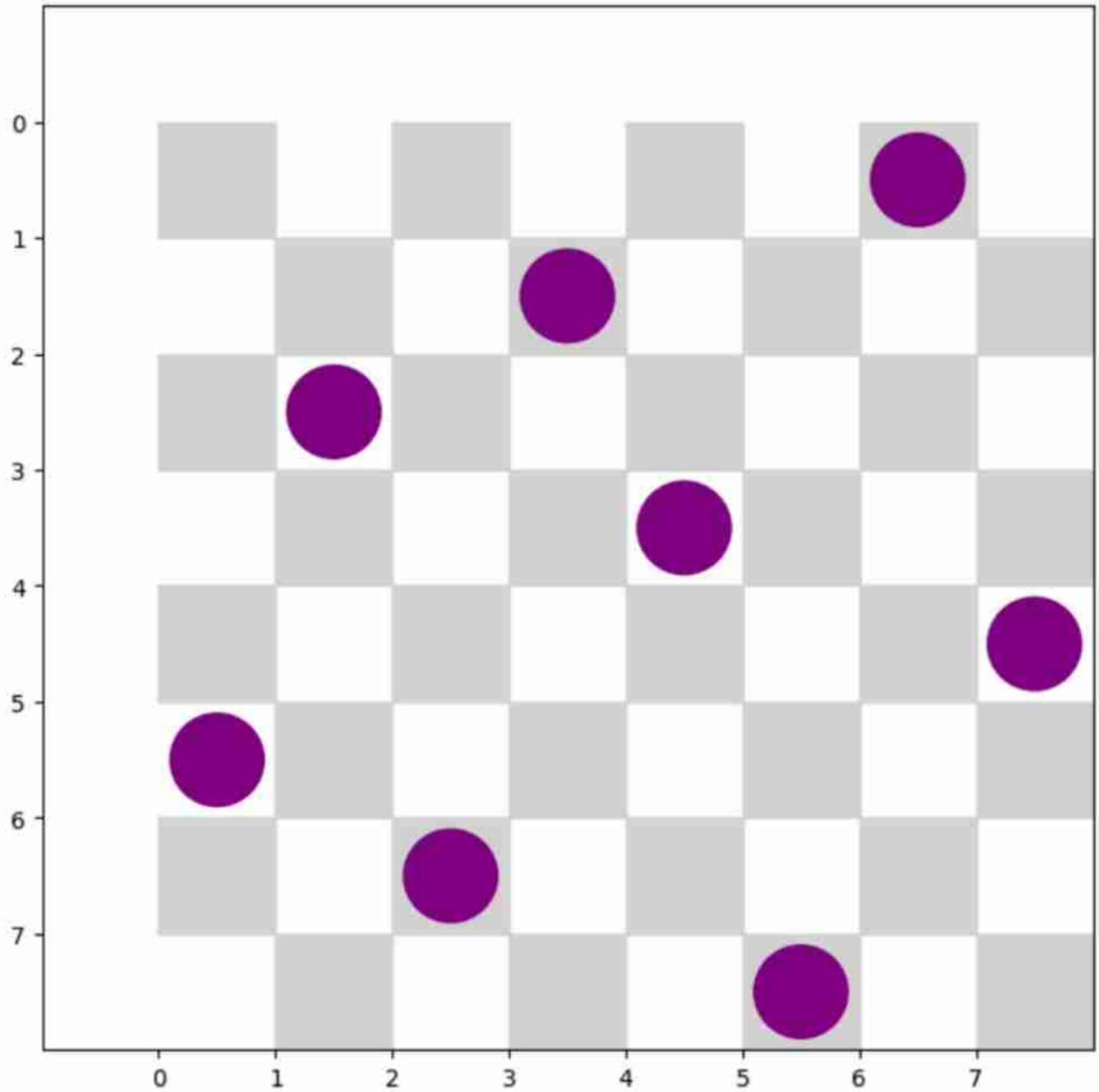
```
import random
import math
import matplotlib.pyplot as plt

# Generate an initial solution with unique columns for each queen
def create_initial_solution(n):
    return random.sample(range(n), n)  # A permutation of column indices (unique columns)
```

```
# Calculate the number of diagonal conflicts
def calculate_fitness(state):
    diagonal_conflicts = 0
    n = len(state)

    for i in range(n):
        for j in range(i + 1, n):
            if abs(state[i] - state[j]) == abs(i - j):  # Check if they are on the same diagonal
                diagonal_conflicts += 1

    return diagonal_conflicts

# Generate a neighboring solution by swapping two columns
def random_neighbor(state):
    neighbor = state[:]
    i, j = random.sample(range(len(state)), 2)  # Pick two different rows to swap
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

# Simulated Annealing Algorithm
def simulated_annealing(n, initial_temp=1000, cooling_rate=0.95, max_iterations=1000):
    current_solution = create_initial_solution(n)
    current_fitness = calculate_fitness(current_solution)
    best_solution = current_solution
    best_fitness = current_fitness
    temperature = initial_temp

    for iteration in range(max_iterations):
        neighbor = random_neighbor(current_solution)
        neighbor_fitness = calculate_fitness(neighbor)

        fitness_diff = neighbor_fitness - current_fitness

        # Accept the neighbor if it improves the solution or based on the annealing probability
        if fitness_diff < 0 or random.uniform(0, 1) < math.exp(-fitness_diff / temperature):
            current_solution = neighbor
            current_fitness = neighbor_fitness

            # Update the best solution if the current one is better
            if current_fitness < best_fitness:
                best_solution = current_solution
                best_fitness = current_fitness

        # Cool down the temperature
        temperature *= cooling_rate
```

```
    return best_solution, best_fitness

# Visualize the chessboard and queens
def plot_solution(solution):
    n = len(solution)
    plt.figure(figsize=(n, n))
    plt.xlim(-1, n)
    plt.ylim(-1, n)

    # Draw the chessboard
    for i in range(n):
        for j in range(n):
            if (i + j) % 2 == 0:
                plt.gca().add_patch(plt.Rectangle((j, i), 1, 1, color='lightgrey'))

    # Place the queens
    for col, row in enumerate(solution):
        plt.gca().add_patch(plt.Circle((col + 0.5, row + 0.5), 0.4, color='purple'))

    plt.xticks(range(n))
    plt.yticks(range(n))
    plt.gca().invert_yaxis()
    plt.grid(False)
    plt.show()

# Parameters
n = 8  # Number of queens
best_solution, best_fitness = simulated_annealing(n)

# Output results
print(f"Best state (Queen positions): {best_solution}, Number of conflicts: {best_fitness}")

# Plot the solution
plot_solution(best_solution)
```

Output:

```
Best state (Queen positions): [5, 2, 6, 1, 3, 7, 0, 4], Number of conflicts: 0
```



Program 6:

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

Algorithm:

12/11/2024

Propositional Logic ~~tmp~~

Q. Implementation of truth-table enumeration algorithm for deciding propositional entailment.

function TT-ENTAILS?(KB, α) return true or false
inputs : KB, the knowledge base, a sentence in
propositional logic
α, a query in propositional logic.

symbols ← a list of propositional symbols in
KB and α

return TT-CHECK-ALL(KB, α, symbols, ~~model~~ { }) ~~returns~~

function TT-CHECK-ALL(KB, α, symbols, model) returns
true or false
if EMPTY? (symbols) then
if PL-TRUE? (KB, model) then return
PL-TRUE? (α, model)
else return true // when KB is false, always
// return true

else do
P ← FIRST(symbols)
rest ← REST (symbols)
return (TT-CHECK-ALL(KB, α, rest, model ∪
{P = true}) and (TT-CHECK-ALL
(KB, α, rest → model ∪ {P = false})

Eg. KB: $(A \vee C) \wedge (B \vee \sim C)$ $\alpha = (A \vee B)$

Check if KB $\models \alpha$ 0 - false 1 - true

A	B	C	(1) $A \vee C$	$\sim C$	(2) $B \vee \sim C$	KB	α
false	false	false	false	true	true	false	false
false	false	true	true	false	false	false	false
false	true	false	false	true	true	false	true
false	true	true	true	false	true	true	true
true	false	false	true	true	true	true	true
true	false	true	true	false	false	false	true
true	true	false	true	true	true	true	true
true	true	true	true	false	true	true	true

12/11/2024

Code:

```
from itertools import product

# Define a function to evaluate a propositional expression
def evaluate(expr, model):
    """
    Evaluates the given expression based on the values in the model.
    """
    for var, val in model.items():
        expr = expr.replace(var, str(val))
    return eval(expr)

# Define the truth-table enumeration algorithm
def truth_table_entails(KB, query, symbols):
    """
    Checks if KB entails query using truth-table enumeration.
    KB: list of propositional expressions (strings)
    query: propositional expression (string)
    symbols: list of symbols (propositions) in the KB and query
    """
    # Generate all possible truth assignments
    assignments = list(product([False, True], repeat=len(symbols)))
```

```
    entailing_models = []

    # Iterate over each assignment to check entailment
    for assignment in assignments:
        model = dict(zip(symbols, assignment))

        # Check if KB is true in this model
        KB_is_true = all(evaluate(expr, model) for expr in KB)

        # If KB is true, check if query is also true
        if KB_is_true:
            query_is_true = evaluate(query, model)
            if query_is_true:
                entailing_models.append(model)  # Store the model
            else:
                return False, []
                # Found a model where KB is true but query is false

    return True, entailing_models  # KB entails query if no counterexample was found

# Get input from the user
symbols = input("Enter the propositions (symbols) separated by spaces: ").split()
KB = []
n = int(input("Enter the number of statements in the knowledge base: "))

for i in range(n):
    expr = input(f"Enter statement {i + 1} in the knowledge base: ")
    KB.append(expr)

query = input("Enter the query: ")

# Check entailment
result, models = truth_table_entails(KB, query, symbols)
if truth_table_entails(KB, query, symbols):
    print("KB entails the query.")
    print("Models where KB entails query:")
    for model in models:
        print(model)
else:
    print("KB does not entail the query.")
```

Output:

```
Enter the propositions (symbols) separated by spaces: A B C
Enter the number of statements in the knowledge base: 2
```

```
Enter statement 1 in the knowledge base: (A or C)
Enter statement 2 in the knowledge base: (B or not C)
Enter the query: A or B
KB entails the query.
Models where KB entails query:
{'A': False, 'B': True, 'C': True}
{'A': True, 'B': False, 'C': False}
{'A': True, 'B': True, 'C': False}
{'A': True, 'B': True, 'C': True}
```

Program 7:

Q. Implement unification in first order logic

Algorithm:

19/11/2024 ~~Lab program~~

Q. Implement unification in first order logic

Algorithm: Unify (Ψ_1, Ψ_2)

1. If Ψ_1 or Ψ_2 is a variable or constant, then:
   a. If Ψ_1 or Ψ_2 are identical, then return NIL.
   b. Else if Ψ_1 is a variable,
      a. then if Ψ_1 occurs in Ψ_2, then return FAILURE
      b. Else return Ψ_1/Ψ_2
   c. Else if Ψ_2 is a variable,
      a. If Ψ_2 occurs in Ψ_1 then return FAILURE,
      b. Else return Ψ_1/Ψ_2
   d. Else return FAILURE

2. If the initial predicate symbol is Ψ_1 and Ψ_2 are not same, then return FAILURE

3. If Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

4. Set substitution set (SUBSET) to NIL.

5. For i=1 to the 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 returns failure.

c. If $S \neq NIL$ then do,

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

b. SUBSET = APPEND(S, SUBSET)

6. Return SUBSET

Eg. $p(x, F(y)) \rightarrow (i)$

$p(a, F(g(x))) \rightarrow (ii)$

~~(i) & (ii) predicate are identical if x is replaced with a in (i)~~

i and ii predicate are identical and no. of arguments are equal

in (i) replace x with a

$p(a, F(y))^{a/x} \longrightarrow (i)$

in (i) replace y with $g(x)$

$p(a, F(g(x)))^{g(x)/y}$

Now (i) and (ii) are same

Eg. $Q(a, g(x,a), f(y)) \rightarrow (i)$

$Q(a, g(f(h), a), x) \rightarrow (ii)$

Replace x in (i) with $f(h)$

$Q(a, g(f(h), a), f(y))^{f(h)/x}$

Replace x in (ii) with $f(y)$

$Q(a, g(f(h), a), f(y))^{f(y)/x}$

~~Now (i) and (ii) are same~~ But the same variable cannot hold 2 values in ψ_1 and ψ_2 hence unification fails.

Q3. $\psi_1 = p(b, x, f(g(z))) \rightarrow ①$

$\psi_2 = p(z, f(y), f(y)) \rightarrow ②$

Q4. $\psi_1 = p(f(a), g(y))$

$\psi_2 = p(x, x)$

This unification is not possible because x cannot take values of f(a) and g(y) with x.

Q5. Replace z in ② with b

$p(b, f(y), f(y))$ b/z

Replace x in ① with f(y)

$p(b, f(y), f(g(z)))$

Replace y in ② with g(z)

$p(b, f(y), f(g(z)))$

② – $p(b, f(y), f(g(z)))$

① – $p(b, f(y), f(g(z)))$

= Unification possible

Eg. $\psi_1 = p(a, x, f(y))$ ~~with~~ $\psi_2 = p(z, b, f(c))$

Replace z with a in ②

$\psi_2 = p(a, b, f(c))$

Replace x with b in ①

$\psi_1 = p(a, b, f(y))$

Replace y with c in ①

$\psi_1 = p(a, b, f(c))$

$\psi_2 = p(a, b, f(c))$

Unification possible.

Code:

```
def occurs_in(var, expr):
    """Check if a variable occurs in an expression."""
    if isinstance(expr, list):
```

```
        return any(occurs_in(var, sub_expr) for sub_expr in expr)
    return var == expr

def unify(x, y, subst):
    """Recursive unification function."""
    if subst is None:
        return None  # Failure
    elif x == y:
        return subst  # No substitution needed
    elif isinstance(x, str) and x.startswith('?'):
        # x is a variable
        if occurs_in(x, y):
            return None  # Failure (occurs check)
        return {**subst, x: y}  # Add substitution
    elif isinstance(y, str) and y.startswith('?'):
        # y is a variable
        if occurs_in(y, x):
            return None  # Failure (occurs check)
        return {**subst, y: x}  # Add substitution
    elif isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        # Unify element-wise
        for xi, yi in zip(x, y):
            subst = unify(xi, yi, subst)
            if subst is None:
                return None
        return subst
    else:
        return None  # Failure


def unification_algorithm(expr1, expr2):
    """Main function to unify two expressions."""
    subst = unify(expr1, expr2, {})
    if subst is None:
        print("Unification Failed")
    else:
        print("Unification Succeeded")
        print("Substitutions:")
        for var, val in subst.items():
            print(f"{var} -> {val}")


def parse_input(expr):
```

```
    """Parses a user input expression into a nested list."""
    try:
        return eval(expr)  # Convert input string to list structure
    except:
        print("Invalid input. Ensure the expression is in proper format.")
        return None

if __name__ == "__main__":
    print("Enter the first expression (e.g., ['P', '?x', 'a']):")
    expr1 = parse_input(input())
    print("Enter the second expression (e.g., ['P', 'b', 'a']):")
    expr2 = parse_input(input())

    if expr1 is not None and expr2 is not None:
        unification_algorithm(expr1, expr2)
    else:
        print("Could not process input. Please try again.")
```

Output:

```
Enter the first expression (e.g., ['P', '?x', 'a']):
['P', 'a', '?X', ['f', '?Y']]
Enter the second expression (e.g., ['P', 'b', 'a']):
['P', '?Z', 'b', ['f', 'c']]
Unification Succeeded
Substitutions:
?Z -> a
?X -> b
?Y -> c
```

Program 8:

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

Algorithm:

26/11/2024 First Order Logic - Forward Chaining

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

Algorithm:

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 $\leftarrow$ { }

for each rule in KB do

$(p_1 \land \dots \land p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)

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

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 or new then

add q' to new

$\phi \leftarrow$ UNIFY(q', α)

if ϕ is not fail then return ϕ

add new to KB

return false

Output expected

Inferred : Weapon (T1)

Inferred : Hostile (A)

Inferred : Sells (Robert, T1, A)

Inferred : Criminal (Robert)

Conclusion : Robert is a Criminal

Code:

```
# Define the knowledge base as a list of rules and facts

class KnowledgeBase:
    def __init__(self):
        self.facts = set()  # Set of known facts
        self.rules = []     # List of rules

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
```

```
        inferred = True
        while inferred:
            inferred = False
            for rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises  # List of conditions
        self.conclusion = conclusion  # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in self.premises):
            if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Inferred: {self.conclusion}")
                return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()

# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")

# Rules based on the problem
# 1. Missile(x) implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p)
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)", "Hostile(A)"],
"Criminal(Robert)"))
```

```python
# Infer new facts based on the rules
kb.infer()

# Check if Robert is a criminal
if "Criminal(Robert)" in kb.facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")
```

Output:

```
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
Conclusion: Robert is a criminal.
```

Program 9:

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

Algorithm:

26/11/2024

Q. Convert a given first order logic statements resolution.

Algorithm:

1. Convert all sentences to CNF
2. Negate conclusion S and convert result to CNF
3. Add negated conclusion S to the premise clauses
4. Repeat until contradiction or no progress is made:
   a. Select 2 clauses (call them parent clauses)
   b. Resolve them together, performing all required unifications
   c. If resolvent is the empty clause, a contradiction has been found (i.e. S follows from the premises)
   d. If not, add resolvent to the premises

If we succeed in Step 4, we have proved the conclusion.

Code:

```
# Step 1: Helper function to parse user inputs into clauses (with '!' for negation)
def parse_clause(clause_input):
    """
    Parses a user input string into a tuple of literals for the clause.
    Replaces '!' with '¬' for negation handling.
    Example: "!Food(x), Likes(John, x)" -> ("¬Food(x)", "Likes(John, x)")
```

```
    """
    return tuple(literal.strip().replace("!", "¬") for literal in clause_input.split(","))

# Step 2: Collect knowledge base (KB) from user
def get_knowledge_base():
    print("Enter the premises of the knowledge base, one by one.")
    print("Use ',' to separate literals in a clause. Use '!' for negation.")
    print("Example: !Food(x), Likes(John, x)")
    print("Type 'done' when finished.")

    kb = []
    while True:
        clause_input = input("Enter a clause (or 'done' to finish): ").strip()
        if clause_input.lower() == "done":
            break
        kb.append(parse_clause(clause_input))

    return kb

# Step 3: Add negated conclusion
def get_negated_conclusion():
    print("\nEnter the conclusion to be proved.")
    print("It will be negated automatically for proof by contradiction.")
    conclusion = input("Enter the conclusion (e.g., Likes(John, Peanuts)): ").strip()
    negated = f"!{conclusion}" if not conclusion.startswith("!") else conclusion[1:]
    return (negated.replace("!", "¬"),)  # Convert '!' to '¬' for consistency

# Step 4: Resolution algorithm
def resolve(clause1, clause2):
    """
    Resolves two clauses and returns the resolvent (new clause).
    If no resolvable literals exist, returns an empty set.
    """
    resolved = set()
    for literal in clause1:
        if "¬" + literal in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove("¬" + literal)
            resolved = temp1.union(temp2)
        elif literal.startswith("¬") and literal[1:] in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove(literal[1:])
            resolved = temp1.union(temp2)
```

```
    return tuple(resolved)

def resolution(kb):
    """
    Runs the resolution algorithm on the knowledge base (KB).
    Returns True if an empty clause is derived (proving the conclusion),
    or False if resolution fails.
    """
    clauses = set(kb)
    new = set()

    while True:
        # Generate all pairs of clauses
        pairs = [(ci, cj) for ci in clauses for cj in clauses if ci != cj]

        for (ci, cj) in pairs:
            resolvent = resolve(ci, cj)
            if not resolvent:  # Empty clause found
                return True
            new.add(resolvent)

        if new.issubset(clauses):  # No new clauses
            return False
        clauses = clauses.union(new)

# Step 5: Main execution
if __name__ == "__main__":
    print("=== Resolution Proof System ===")
    print("Provide the knowledge base and conclusion to prove.")

    # Collect user inputs
    kb = get_knowledge_base()
    negated_conclusion = get_negated_conclusion()
    kb.append(negated_conclusion)

    # Show the knowledge base
    print("\nKnowledge Base (KB):")
    for clause in kb:
        print("  ", " ∨ ".join(clause))  # Join literals with OR for readability

    # Perform resolution
    print("\nStarting resolution process...")
    result = resolution(kb)
    if result:
        print("\nProof complete: The conclusion is TRUE.")
    else:
        print("\nResolution failed: The conclusion could not be proved.")
```

Output:

=== Resolution Proof System ===
Provide the knowledge base and conclusion to prove.
Enter the premises of the knowledge base, one by one.
Use ',' to separate literals in a clause. Use '!' for negation.
Example: !Food(x), Likes(John, x)
Type 'done' when finished.
Enter a clause (or 'done' to finish): !Food(x), Likes(John,x)
Enter a clause (or 'done' to finish): Food(Apple)
Enter a clause (or 'done' to finish): Food(Vegetables)
Enter a clause (or 'done' to finish): !Eats(x,y), !Killed(x), Food(y)
Enter a clause (or 'done' to finish): Eats(Anil, Peanuts)
Enter a clause (or 'done' to finish): !Killed(Anil)
Enter a clause (or 'done' to finish): done

Enter the conclusion to be proved.
It will be negated automatically for proof by contradiction.
Enter the conclusion (e.g., Likes(John, Peanuts)): Likes(John, Peanuts)

Knowledge Base (KB):
  ¬Food(x) ∨ Likes(John ∨ x)
  Food(Apple)
  Food(Vegetables)
  ¬Eats(x ∨ y) ∨ ¬Killed(x) ∨ Food(y)
  Eats(Anil ∨ Peanuts)
  ¬Killed(Anil)
  ¬Likes(John, Peanuts)

Starting resolution process...

Proof complete: The conclusion is TRUE.

Program 10

Q.Implement Alpha-Beta Pruning.

Algorithm:

26/11/2024

Q. Implement Alpha-Beta Pruning

Alpha Beta Pruning Algorithm:

1. Alpha(α) - Beta(β) proposes to compute find the optimal path without looking at every node in the game tree.
2. Max contains Alpha(α) and min contains Beta(β) bound during the calculation.
3. In both min and max node we return when $\alpha \geq \beta$ which compares with its parent node only.
4. Both minimax and Alpha(α) - Beta(β) cut-off give same path.
5. Alpha(α) - Beta(β) gives optimal solution as it takes less time to get the value for the root node.

Code:

```
# Python3 program to demonstrate
# working of Alpha-Beta Pruning

# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
                    values, alpha, beta):
```

```
    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                          False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

    else:
        best = MAX

        # Recur for left and
        # right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                            True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

# Driver Code
if __name__ == "__main__":

    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output:
The optimal value is : 5