Program 1:
Implement and Demonstrate Depth First Search Algorithm on Water Jug Problem

```
class State:
    def __init__(self, jug1, jug2):
        # Initialize the state of the jugs
        self.jug1 = jug1  # Amount of water in jug1
        self.jug2 = jug2  # Amount of water in jug2

    def __eq__(self, other):
        # Define equality of states
        return self.jug1 == other.jug1 and self.jug2 == other.jug2

    def __hash__(self):
        # Define hash function for states
        return hash((self.jug1, self.jug2))

def dfs(start, target, visited):
    # Check if a state is the goal state
    if start == target:
        return [start]  # If the current state is the target state, return it

    visited.add(start)  # Add the current state to visited states

    for next_state in get_successors(start):  # For each possible next state
        if next_state not in visited:  # If the next state has not been visited
            path = dfs(next_state, target, visited)  # Recursively call DFS on the next state
            if path:  # If a solution path is found
                return [start] + path  # Return the current state and the solution path
    return None  # If no solution path is found, return None

def get_successors(state):
    # Generate successor states from a given state
    successors = []  # List to store successor states

    # Fill jug1 (fill jug1 to its capacity, which is 3)
    if state.jug1 < 3:
        successors.append(State(jug1=3, jug2=state.jug2))

    # Fill jug2 (fill jug2 to its capacity, which is 4)
    if state.jug2 < 4:
        successors.append(State(jug1=state.jug1, jug2=4))

    # Empty jug1 (empty jug1 to 0)
    if state.jug1 > 0:
        successors.append(State(jug1=0, jug2=state.jug2))

    # Empty jug2 (empty jug2 to 0)
    if state.jug2 > 0:
        successors.append(State(jug1=state.jug1, jug2=0))

    # Pour jug1 to jug2
    pour_amount = min(state.jug1, 4 - state.jug2)  # Amount that can be poured from jug1 to
jug2
    if pour_amount > 0:
```

```
        successors.append(State(jug1=state.jug1 - pour_amount, jug2=state.jug2 +
pour_amount))

    # Pour jug2 to jug1
    pour_amount = min(state.jug2, 3 - state.jug1)  # Amount that can be poured from jug2 to
jug1
    if pour_amount > 0:
        successors.append(State(jug1=state.jug1 + pour_amount, jug2=state.jug2 -
pour_amount))

    return successors

def print_solution(path):
    # Print the solution path
    print("Solution:")
    for i, state in enumerate(path):
        print(f"Step {i}: Jug1={state.jug1}, Jug2={state.jug2}")

def main():
    # Initial state of the jugs
    start_state = State(jug1=0, jug2=0)

    # Target state we want to achieve (2 units in jug1 and 0 units in jug2)
    target_state = State(jug1=2, jug2=0)

    # Set to store visited states
    visited = set()

    # Find solution using DFS
    path = dfs(start_state, target_state, visited)

    if path:  # If solution path is found
        print_solution(path)  # Print the solution path
    else:  # If no solution path is found
        print("No solution found.")

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

File Edit View Run Kernel Settings Help

Code

```python
        print_solution(path)  # Print the solution path
    else:  # If no solution path is found
        print("No solution found.")

if __name__ == "__main__":
    main()

```

```
Solution:
Step 0: Jug1=0, Jug2=0
Step 1: Jug1=3, Jug2=0
Step 2: Jug1=3, Jug2=4
Step 3: Jug1=0, Jug2=4
Step 4: Jug1=3, Jug2=1
Step 5: Jug1=0, Jug2=1
Step 6: Jug1=1, Jug2=0
Step 7: Jug1=1, Jug2=4
Step 8: Jug1=3, Jug2=2
Step 9: Jug1=0, Jug2=2
Step 10: Jug1=2, Jug2=0
```

[ ]:

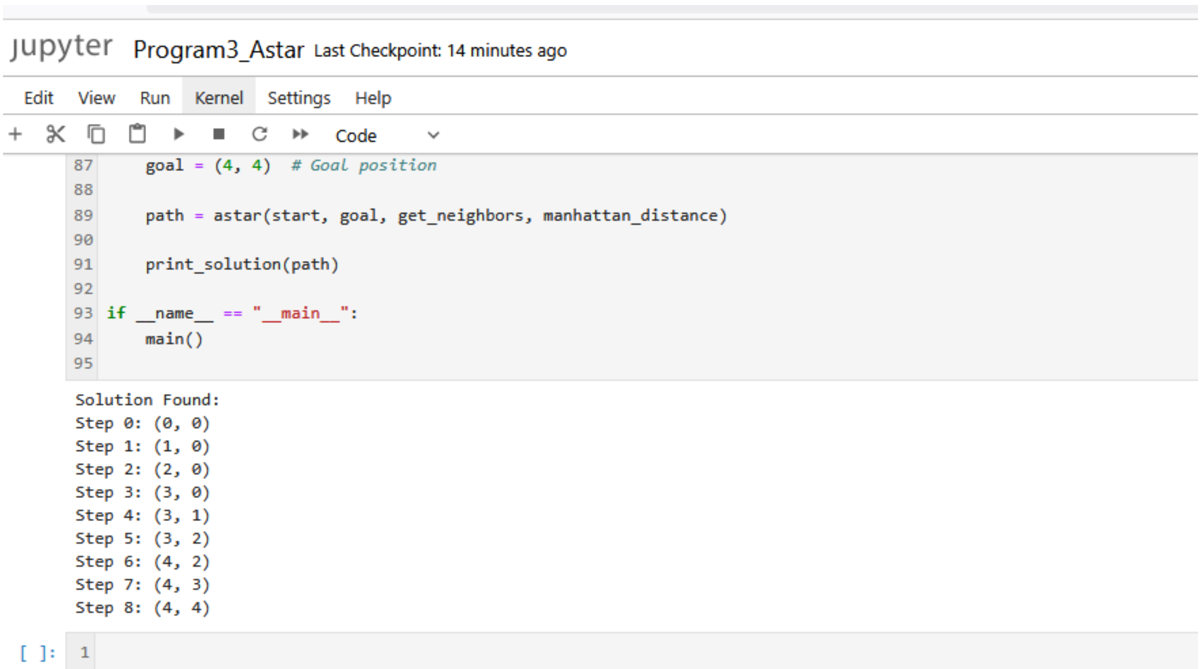
```python

```

Program 2: Implement and Demonstrate Best First Search Algorithm on Missionaries-Cannibals Problems using Python

Program-3: Implement A* Search algorithm

```
import heapq

class Node:
    """Class representing a node in the search tree."""
    def __init__(self, state, parent=None):
        self.state = state  # (x, y) position
        self.parent = parent  # Parent node
        self.g = 0  # Cost from start node to current node
        self.h = 0  # Estimated cost from current node to goal node

    def __lt__(self, other):
        """Compare nodes based on their f-score (g + h)."""
        return (self.g + self.h) < (other.g + other.h)

def astar(start, goal, get_neighbors, heuristic):
    """A* Search Algorithm"""
    open_list = []  # Priority queue for nodes to be evaluated
    closed_set = set()  # Set of evaluated nodes

    start_node = Node(start)
    start_node.h = heuristic(start, goal)
    heapq.heappush(open_list, start_node)  # Add start node to open list

    while open_list:
        current_node = heapq.heappop(open_list)  # Get node with lowest f-score

        if current_node.state == goal:
            # Goal reached, construct path
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]  # Reverse to get path from start to goal

        closed_set.add(current_node.state)

        for neighbor in get_neighbors(current_node.state):
            if neighbor in closed_set:
                continue  # Skip already evaluated nodes

            tentative_g = current_node.g + 1  # Assuming uniform cost

            neighbor_node = Node(neighbor, current_node)
            neighbor_node.g = tentative_g
            neighbor_node.h = heuristic(neighbor, goal)

            # Ensure that the node is updated if a better path is found
            in_open_list = False
            for node in open_list:
                if node.state == neighbor:
                    in_open_list = True
                    if tentative_g < node.g:
                        node.g = tentative_g
                        node.parent = current_node
                    break

            if not in_open_list:
                heapq.heappush(open_list, neighbor_node)
```

```
    return None  # No path found

def manhattan_distance(state, goal):
    """Heuristic function: Manhattan distance."""
    return abs(state[0] - goal[0]) + abs(state[1] - goal[1])

def get_neighbors(state):
    """Generate valid neighboring positions on a 5x5 grid."""
    x, y = state
    neighbors = []
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:  # Up, Down, Left, Right
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 5 and 0 <= new_y < 5:  # Keep within 5x5 grid bounds
            neighbors.append((new_x, new_y))
    return neighbors

def print_solution(path):
    """Print the solution path."""
    if path:
        print("Solution Found:")
        for step, state in enumerate(path):
            print(f"Step {step}: {state}")
    else:
        print("No solution found.")

def main():
    start = (0, 0)  # Starting position
    goal = (4, 4)  # Goal position

    path = astar(start, goal, get_neighbors, manhattan_distance)

    print_solution(path)

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



Program 4: Implement AO* Search algorithm

```
import heapq

class Node:
    """A class representing a node in the search tree."""
    def __init__(self, state, parent=None):
        self.state = state  # Current state of the node
        self.parent = parent  # Parent node
        self.g = 0  # Cost from start node to current node
        self.h = 0  # Heuristic estimate from current node to goal node
        self.f = 0  # Total estimated cost (g + h)

    def __lt__(self, other):
        """Comparison function for priority queue (min heap)."""
        return self.f < other.f

def ao_star(start, goal, get_neighbors, heuristic):
    """AO* search algorithm."""
    open_list = []  # Priority queue for nodes to be evaluated
    open_dict = {}  # Dictionary to track nodes in open_list
    closed_set = set()  # Set of nodes already evaluated

    # Initialize start node
    start_node = Node(state=start)
    start_node.h = heuristic(start, goal)
    start_node.f = start_node.g + start_node.h
    heapq.heappush(open_list, start_node)
    open_dict[start] = start_node

    while open_list:
        current_node = heapq.heappop(open_list)
        open_dict.pop(current_node.state, None)  # Remove from tracking

        if current_node.state == goal:
            # Goal reached, reconstruct the path
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            path.reverse()
            return path

        closed_set.add(current_node.state)

        for neighbor_state in get_neighbors(current_node.state):
            if neighbor_state in closed_set:
                continue  # Skip already evaluated nodes

            # Compute tentative cost
            tentative_g = current_node.g + 1  # Assuming uniform step cost

            if neighbor_state in open_dict:
                neighbor_node = open_dict[neighbor_state]
                if tentative_g < neighbor_node.g:
                    neighbor_node.g = tentative_g
                    neighbor_node.f = neighbor_node.g + neighbor_node.h
                    neighbor_node.parent = current_node
                    heapq.heapify(open_list)  # Re-sort priority queue
            else:
```

```python
                # Create new node
                neighbor_node = Node(state=neighbor_state, parent=current_node)
                neighbor_node.g = tentative_g
                neighbor_node.h = heuristic(neighbor_state, goal)
                neighbor_node.f = neighbor_node.g + neighbor_node.h
                heapq.heappush(open_list, neighbor_node)
                open_dict[neighbor_state] = neighbor_node

    return None  # No path found

def manhattan_distance(state, goal):
    """Heuristic function: Manhattan distance."""
    return abs(state[0] - goal[0]) + abs(state[1] - goal[1])

def get_neighbors(state):
    """Get neighboring states (moving up, down, left, right in a 5x5 grid)."""
    x, y = state
    neighbors = []
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 5 and 0 <= new_y < 5:  # Grid boundaries
            neighbors.append((new_x, new_y))
    return neighbors

def print_solution(path):
    """Print the solution path."""
    print("Solution Path:")
    for i, state in enumerate(path):
        print(f"Step {i}: {state}")

def main():
    """Example problem: Find path in a 5x5 grid."""
    start = (0, 0)
    goal = (4, 4)
    path = ao_star(start, goal, get_neighbors, manhattan_distance)
    if path:
        print_solution(path)
    else:
        print("No solution found.")

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

Program3_Astar

localhost:8890/notebooks/Program3_Astar.ipynb

jupyter Program3_Astar Last Checkpoint: 19 minutes ago

File Edit View Run Kernel Settings Help

Code

```python
    path = ao_star(start, goal, get_neighbors, manhattan_distance)
    if path:
        print_solution(path)
    else:
        print("No solution found.")

if __name__ == "__main__":
    main()

```

```
Solution Path:
Step 0: (0, 0)
Step 1: (1, 0)
Step 2: (2, 0)
Step 3: (3, 0)
Step 4: (3, 1)
Step 5: (3, 2)
Step 6: (4, 2)
Step 7: (4, 3)
Step 8: (4, 4)
```

[ ]:

Program 5: Solve 8-Queens Problem with suitable assumptions

```
def is_safe(board, row, col):
    """Check if a queen can be placed at (row, col)"""
    for i in range(row):
        # Check same column
        if board[i] == col:
            return False
        # Check upper left diagonal
        if board[i] == col - (row - i):
            return False
        # Check upper right diagonal
        if board[i] == col + (row - i):
            return False
    return True

def solve_queens(board, row, solution_count):
    """Backtracking function to place queens on the board."""
    if row >= 8:  # Base case: All queens are placed
        solution_count[0] += 1  # Increment solution count
        if solution_count[0] == 1:  # Print only the first solution
            print_board(board)
        return

    for col in range(8):
        if is_safe(board, row, col):
            board[row] = col  # Place queen
            solve_queens(board, row + 1, solution_count)  # Recursive call for next row
            board[row] = -1  # Backtrack if needed

def print_board(board):
    """Print the chessboard with queens placed."""
    for i in range(8):
        for j in range(8):
            if board[i] == j:
                print("Q", end=" ")  # Print queen
            else:
                print(".", end=" ")  # Print empty cell
        print()
    print()  # Add space after solution

def main():
    """Main function to solve the 8-Queens problem."""
    board = [-1] * 8  # Initialize board with empty (-1)
    solution_count = [0]  # List to hold solution count
    solve_queens(board, 0, solution_count)
    print(f"Total number of solutions: {solution_count[0]}")

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

Code

```python
    """Main function to solve the 8-Queens proble
    board = [-1] * 8  # Initialize board with emp
    solution_count = [0]  # List to hold solutior
    solve_queens(board, 0, solution_count)
    print(f"Total number of solutions: {solution_

if __name__ == "__main__":
    main()

```

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

Total number of solutions: 92
```