Prog 1:

```
class WaterJugState:
    def __init__(self, jug1, jug2):
        self.jug1 = jug1
        self.jug2 = jug2

    def __eq__(self, other):
        return self.jug1 == other.jug1 and self.jug2 == other.jug2

    def __hash__(self):
        return hash((self.jug1, self.jug2))

def dfs(current_state, visited, jug1_capacity, jug2_capacity, target_volume):
    if current_state.jug1 == target_volume or current_state.jug2 == target_volume:
        if current_state.jug1 == target_volume :
            print("Jug 1 now has", target_volume, "liters.")
        else:
            print("Jug 2 now has", target_volume, "liters.")
        return True

    visited.add(current_state)

    # Define all possible operations: (action, from_jug, to_jug)
    operations = [
        ('Fill Jug 1', jug1_capacity, current_state.jug2),
        ('Fill Jug 2', current_state.jug1, jug2_capacity),
        ('Empty Jug 1', 0, current_state.jug2),
        ('Empty Jug 2', current_state.jug1, 0),
        ('Pour Jug 1 to Jug 2',
            max(0, current_state.jug1 + current_state.jug2 - jug2_capacity),
            min(jug2_capacity, current_state.jug1 + current_state.jug2)),
```

```
        ('Pour Jug 2 to Jug 1',
         min(jug1_capacity, current_state.jug1 + current_state.jug2),
         max(0, current_state.jug1 + current_state.jug2 - jug1_capacity))
    ]

    for operation in operations:
        action, new_jug1, new_jug2 = operation
        new_state = WaterJugState(new_jug1, new_jug2)

        if new_state not in visited:
            print(f"Trying: {action} => ({new_jug1}, {new_jug2})")
            if dfs(new_state, visited, jug1_capacity, jug2_capacity, target_volume):
                return True
#       print(new_jug1, new_jug2)
    return False

def solve_water_jug_problem(jug1_capacity, jug2_capacity, target_volume):
    initial_state = WaterJugState(0, 0)
    visited = set()

    if dfs(initial_state, visited, jug1_capacity, jug2_capacity, target_volume):
        print("Solution found!")
#       print(jug1_capacity, jug2_capacity)
    else:
        print("Solution not possible.")

# Example usage:
jug1_capacity = int(input("Enter Jug 1 capacity : "))
jug2_capacity = int(input("Enter Jug 1 capacity : "))
target_volume = int(input("Enter Target Volume : "))
```

```
print(f"Solving Water Jug Problem with capacities ({jug1_capacity}, {jug2_capacity}) to measure
{target_volume} liters.")

solve_water_jug_problem(jug1_capacity, jug2_capacity, target_volume)
```

Prog 2:

```
from queue import PriorityQueue

# Define the state class for the Missionaries and Cannibals Problem
class State:
    def __init__(self, left_m, left_c, boat, right_m, right_c):
        self.left_m = left_m  # Number of missionaries on the left bank
        self.left_c = left_c  # Number of cannibals on the left bank
        self.boat = boat      # 1 if boat is on the left bank, 0 if on the right bank
        self.right_m = right_m  # Number of missionaries on the right bank
        self.right_c = right_c  # Number of cannibals on the right bank

    def is_valid(self):
        # Check if the state is valid (no missionaries eaten on either bank)
        if self.left_m < 0 or self.left_c < 0 or self.right_m < 0 or self.right_c < 0:
            return False
        if self.left_m > 0 and self.left_c > self.left_m:
            return False
        if self.right_m > 0 and self.right_c > self.right_m:
            return False
        return True

    def is_goal(self):
        # Check if the state is the goal state (all missionaries and cannibals on the right bank)
        return self.left_m == 0 and self.left_c == 0
```

```
    def __lt__(self, other):
        # Define less-than operator for PriorityQueue comparison (used in Best-First Search)
        return False

    def __eq__(self, other):
        # Define equality operator for comparing states
        return self.left_m == other.left_m and self.left_c == other.left_c \
            and self.boat == other.boat and self.right_m == other.right_m \
            and self.right_c == other.right_c

    def __hash__(self):
        # Define hash function for storing states in a set
        return hash((self.left_m, self.left_c, self.boat, self.right_m, self.right_c))

def successors(state):
    # Generate all valid successor states from the current state
    succ_states = []
    if state.boat == 1:  # Boat is on the left bank
        for m in range(3):
            for c in range(3):
                if 1 <= m + c <= 2:  # Boat capacity is 2
                    new_state = State(state.left_m - m, state.left_c - c, 0,
                                      state.right_m + m, state.right_c + c)
                    if new_state.is_valid():
                        succ_states.append(new_state)
    else:  # Boat is on the right bank
        for m in range(3):
            for c in range(3):
                if 1 <= m + c <= 2:  # Boat capacity is 2
                    new_state = State(state.left_m + m, state.left_c + c, 1,
                                      state.right_m - m, state.right_c - c)
```

```
            if new_state.is_valid():
                succ_states.append(new_state)
    return succ_states

def best_first_search():
    start_state = State(3, 3, 1, 0, 0)
    goal_state = State(0, 0, 0, 3, 3)

    frontier = PriorityQueue()
    frontier.put((0, start_state))  # Priority queue with (cost, state)
    came_from = {}
    cost_so_far = {}
    came_from[start_state] = None
    cost_so_far[start_state] = 0

    while not frontier.empty():
        current_cost, current_state = frontier.get()

        if current_state == goal_state:
            # Reconstruct the path from start_state to goal_state
            path = []
            while current_state is not None:
                path.append(current_state)
                current_state = came_from[current_state]
            path.reverse()
            return path

        for next_state in successors(current_state):
            new_cost = cost_so_far[current_state] + 1  # Uniform cost of 1 for each move

            if next_state not in cost_so_far or new_cost < cost_so_far[next_state]:
```

```python
            cost_so_far[next_state] = new_cost
            priority = new_cost  # Best-First Search uses cost as priority
            frontier.put((priority, next_state))
            came_from[next_state] = current_state

    return None  # No path found

def print_solution(path):
    if path is None:
        print("No solution found.")
    else:
        print("Solution found!")
        for i, state in enumerate(path):
            print(f"Step {i}:")
            print(f"Left Bank: {state.left_m} missionaries, {state.left_c} cannibals")
            print(f"Boat is {'on the left' if state.boat == 1 else 'on the right'} bank")
            print(f"Right Bank: {state.right_m} missionaries, {state.right_c} cannibals")
            print("------------")

# Main function to run the Best-First Search and print the solution
if __name__ == "__main__":
    solution_path = best_first_search()
    print_solution(solution_path)
```

Prog 3:

```python
import heapq

class Node:
    def __init__(self, state, parent=None, action=None, cost=0, heuristic=0):
        self.state = state      # Current state in the search space
```

```
        self.parent = parent    # Parent node
        self.action = action    # Action that led to this node from the parent node
        self.cost = cost        # Cost to reach this node from the start node
        self.heuristic = heuristic  # Heuristic estimate of the cost to reach the goal

    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def parse_graph_input():
    graph = {}
    num_edges = int(input("Enter the number of edges: "))
    for _ in range(num_edges):
        u, v, cost = input("Enter an edge (format: u v cost): ").split()
        cost = int(cost)
        if u not in graph:
            graph[u] = []
        if v not in graph:
            graph[v] = []
        graph[u].append((v, cost))
        graph[v].append((u, cost))
    return graph

def astar_search(start_state, goal_test, successors, heuristic):
    # Priority queue to store nodes ordered by f = g + h
    frontier = []
    heapq.heappush(frontier, Node(start_state, None, None, 0, heuristic(start_state)))
    explored = set()

    while frontier:
        current_node = heapq.heappop(frontier)
        current_state = current_node.state
```

```
        if goal_test(current_state):
            # Reconstruct the path from the goal node to the start node
            path = []
            while current_node.parent is not None:
                path.append((current_node.action, current_node.state))
                current_node = current_node.parent
            path.reverse()
            return path

        explored.add(current_state)

        # Generate successors for the current state using the `successors` function
        for action, successor_state, step_cost in successors(current_state):
            if successor_state not in explored:
                new_cost = current_node.cost + step_cost
                new_node = Node(successor_state, current_node, action, new_cost,
heuristic(successor_state))
                heapq.heappush(frontier, new_node)

    return None  # No path found

if __name__ == "__main__":
    # Get user input to define the graph
    print("Define the graph:")
    graph = parse_graph_input()

    start_state = input("Enter the start state: ")
    goal_state = input("Enter the goal state: ")

    def goal_test(state):
```

```
        return state == goal_state

    def successors(state):
        # Generate successor states from the current state based on the graph
        successors_list = []
        for neighbor, cost in graph.get(state, []):
            action = f"Move to {neighbor}"   # Default action (e.g., "Move to B")
            successor_state = neighbor
            step_cost = cost
            successors_list.append((action, successor_state, step_cost))
        return successors_list

    def heuristic(state):
        # Define a simple heuristic function (e.g., straight-line distance)
        heuristic_values = {key: abs(ord(key) - ord(goal_state)) for key in graph.keys()}
        return heuristic_values.get(state, float('inf'))   # Default to infinity if state not found

    # Perform A* search using custom successors function
    path = astar_search(start_state, goal_test, successors, heuristic)

    # Print the resulting path found by A* search
    if path:
        print("Path found:")
        for action, state in path:
            print(f"Action: {action}, State: {state}")
    else:
        print("No path found.")
```

Prog 4:

```
import heapq
```

```
class Node:
   def __init__(self, state, parent=None, action=None, cost=0, heuristic=0):
      self.state = state      # Current state in the search space
      self.parent = parent    # Parent node
      self.action = action    # Action that led to this node from the parent node
      self.cost = cost        # Cost to reach this node from the start node
      self.heuristic = heuristic  # Heuristic estimate of the cost to reach the goal

   def __lt__(self, other):
      return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def parse_graph_input():
   graph = {}
   num_edges = int(input("Enter the number of edges: "))
   for _ in range(num_edges):
      u, v, cost = input("Enter an edge (format: u v cost): ").split()
      cost = float(cost)
      if u not in graph:
         graph[u] = []
      if v not in graph:
         graph[v] = []
      graph[u].append((v, cost))
   return graph

def ao_star_search(start_state, goal_state, graph):
   frontier = []
   heapq.heappush(frontier, Node(start_state, None, None, 0, heuristic(start_state, goal_state)))
   explored = {}

   while frontier:
```

```
        current_node = heapq.heappop(frontier)
        current_state = current_node.state

        if current_state == goal_state:
            # Reconstruct the path from the goal node to the start node
            path = []
            while current_node.parent is not None:
                path.append((current_node.action, current_node.state))
                current_node = current_node.parent
            path.reverse()
            return path

        if current_state not in explored or current_node.cost < explored[current_state]:
            explored[current_state] = current_node.cost

            for neighbor, step_cost in graph.get(current_state, []):
                new_cost = current_node.cost + step_cost
                new_node = Node(neighbor, current_node, f"Move to {neighbor}", new_cost,
heuristic(neighbor, goal_state))
                heapq.heappush(frontier, new_node)

    return None # No path found

def heuristic(state, goal_state):
    # Simple heuristic function (e.g., straight-line distance)
    heuristic_values = {'A': 5, 'B': 3, 'C': 2, 'D': 1, 'E': 2, 'G': 0} # Custom heuristic values based on
problem domain
    return heuristic_values.get(state, float('inf')) # Default to infinity if state not found

if __name__ == "__main__":
    # Get user input to define the graph
    print("Define the graph:")
```

```
    graph = parse_graph_input()

    start_state = input("Enter the start state: ")
    goal_state = input("Enter the goal state: ")

    # Perform AO* search using the defined graph, start state, and goal state
    path = ao_star_search(start_state, goal_state, graph)

    # Print the resulting path found by AO* search
    if path:
        print("Path found:")
        for action, state in path:
            print(f"Action: {action}, State: {state}")
    else:
        print("No path found.")
```

Prog 5:

```
def is_safe(board, row, col):
    """ Check if it's safe to place a queen at board[row][col] """
    # Check column
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper diagonal on left side
    i, j = row, col
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
        i -= 1
        j -= 1
```

```
    # Check upper diagonal on right side
    i, j = row, col
    while i >= 0 and j < len(board):
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1

    return True

def solve_queens(board, row):
    """ Recursively solve the 8-Queens Problem using backtracking """
    n = len(board)

    # Base case: If all queens are placed, return True
    if row >= n:
        return True

    for col in range(n):
        if is_safe(board, row, col):
            board[row][col] = 1  # Place the queen

            # Recur to place the rest of the queens
            if solve_queens(board, row + 1):
                return True

            # If placing queen at board[row][col] doesn't lead to a solution, backtrack
            board[row][col] = 0  # Backtrack

    return False
```

```
def print_board(board):
    """ Print the board configuration """
    n = len(board)
    for i in range(n):
        for j in range(n):
            print(board[i][j], end=" ")
        print()

def solve_8queens():
    """ Solve the 8-Queens Problem and print the solution """
    n = 8  # Size of the chessboard (8x8)
    board = [[0] * n for _ in range(n)]  # Initialize empty board

    if solve_queens(board, 0):
        print("Solution found:")
        print_board(board)
    else:
        print("No solution exists.")

# Call the function to solve the 8-Queens Problem
solve_8queens()
```

Prog 6:

```
import sys

def nearest_neighbor_tsp(distances):
    num_cities = len(distances)
```

```
    # Start from the first city (arbitrary choice)
    tour = [0]  # Store the tour as a list of city indices
    visited = set([0])  # Track visited cities

    current_city = 0
    total_distance = 0

    while len(visited) < num_cities:
        nearest_city = None
        min_distance = sys.maxsize

        # Find the nearest unvisited city
        for next_city in range(num_cities):
            if next_city not in visited and distances[current_city][next_city] < min_distance:
                nearest_city = next_city
                min_distance = distances[current_city][next_city]

        # Move to the nearest city
        tour.append(nearest_city)
        visited.add(nearest_city)
        total_distance += min_distance
        current_city = nearest_city

    # Complete the tour by returning to the starting city
    tour.append(0)
    total_distance += distances[current_city][0]

    return tour, total_distance

# Example usage:
if __name__ == "__main__":
```

```
    # Example distance matrix (symmetric, square matrix)
#    distances = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
    distances = [[ 0,  4,  8,  9, 12], [ 4,  0,  6,  8,  9], [ 8,  6,  0, 10, 11], [ 9,  8, 10,  0,  7], [12,  9, 11,  7,  0]]
    # Run nearest neighbor TSP algorithm
    tour, total_distance = nearest_neighbor_tsp(distances)

    # Print the tour and total distance
    print("Nearest Neighbor TSP Tour:", tour)
    print("Total Distance:", total_distance)
```

Prog 7:

```
def forward_chaining(rules, facts, goal):
    inferred_facts = set(facts)
    new_facts = True

    while new_facts:
        new_facts = False

        for rule in rules:
            condition, result = rule

            if all(cond in inferred_facts for cond in condition) and result not in inferred_facts:
                inferred_facts.add(result)
                new_facts = True

                if result == goal:
                    return True

    return False
```

```
def backward_chaining(rules, facts, goal):
    def ask(query):
        if query in facts:
            return True

        for rule in rules:
            condition, result = rule
            if result == query and all(ask(cond) for cond in condition):
                return True

        return False

    return ask(goal)



# Define the rules and facts for the animal classification problem
rules = [
    (['hair', 'live young'], 'mammal'),
    (['feathers', 'fly'], 'bird')
]

facts = ['hair', 'live young']
goal = 'mammal'


# Use forward chaining to determine if a cat is classified as a mammal
is_mammal = forward_chaining(rules, facts, goal)

if is_mammal:
```

```python
    print("The cat is classified as a mammal.")
else:
    print("The cat is not classified as a mammal.")

facts = ['feathers', 'fly']
goal = 'bird'

# Use backward chaining to determine if a pigeon is classified as a bird
is_bird = backward_chaining(rules, facts, goal)

if is_bird:
    print("The pigeon is classified as a bird.")
else:
    print("The pigeon is not classified as a bird.")
```

Prog 8:

```python
def negate_literal(literal):
    """ Negate a literal by adding or removing the negation symbol '~' """
    if literal.startswith('~'):
        return literal[1:]  # Remove negation
    else:
        return '~' + literal  # Add negation

def resolve(clause1, clause2):
    """ Resolve two clauses to derive a new clause """
    new_clause = []
    resolved = False

    # Copy literals from both clauses
    for literal in clause1:
```

```
      if negate_literal(literal) in clause2:
        resolved = True
      else:
        new_clause.append(literal)

    for literal in clause2:
      if negate_literal(literal) not in clause1:
        new_clause.append(literal)

    if resolved:
      return new_clause
    else:
      return None  # No resolution possible

def resolution(propositional_kb, query):
    """ Use resolution to prove or disprove a query using propositional logic """
    kb = propositional_kb[:]
    kb.append(negate_literal(query))  # Add negated query to knowledge base

    while True:
      new_clauses = []
      n = len(kb)
      resolved_pairs = set()  # Track resolved pairs to avoid redundant resolutions

      for i in range(n):
        for j in range(i + 1, n):
          clause1 = kb[i]
          clause2 = kb[j]

          if (clause1, clause2) not in resolved_pairs:
            resolved_pairs.add((clause1, clause2))
```

```
                resolvent = resolve(clause1, clause2)

                if resolvent is None:
                    continue  # No resolution possible for these clauses

                if len(resolvent) == 0:
                    return True  # Empty clause (contradiction), query is proved

                if resolvent not in new_clauses:
                    new_clauses.append(resolvent)

        if all(clause in kb for clause in new_clauses):
            return False  # No new clauses added, query cannot be proven

        kb.extend(new_clauses)  # Add new clauses to the knowledge base

# Example usage:
if __name__ == "__main__":
    # Example propositional knowledge base (list of clauses)
    propositional_kb = [
        ['~P', 'Q'],
        ['P', '~Q', 'R'],
        ['~R', 'S']
    ]

    # Example query to prove/disprove using resolution
    query = 'S'

    # Use resolution to prove or disprove the query
    result = resolution(propositional_kb, query)
```

```
    if result:
        print(f"The query '{query}' is PROVED.")
    else:
        print(f"The query '{query}' is DISPROVED.")
```

Prog 9:

```
def print_board(board):
    """ Print the current state of the Tic-Tac-Toe board """
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board, player):
    """ Check if the specified player has won the game """
    for row in board:
        if all(cell == player for cell in row):
            return True
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2-i] == player for i in range(3)):
        return True
    return False

def is_full(board):
    """ Check if the board is completely filled """
    return all(cell != ' ' for row in board for cell in row)
```

```python
def tic_tac_toe():
    """ Main function to run the Tic-Tac-Toe game """
    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X'

    while True:
        print_board(board)
        print(f"Player {current_player}'s turn.")
        row = int(input("Enter row (1-3): "))
        col = int(input("Enter column (1-3): "))
        row -= 1
        col -= 1

        if board[row][col] == ' ':
            board[row][col] = current_player
        else:
            print("Invalid move! Try again.")
            continue

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

        # Check if the board is full (tie game)
        if is_full(board):
            print_board(board)
            print("It's a tie!")
            break
```

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


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

Prog 10:

```
import random
import sys

# Define some rules and responses
rules = {
    "hi": ["Hello!", "Hi there!", "Hey!"],
    "how are you": ["I'm good, thanks!", "I'm doing well, how about you?", "All good!"],
    "bye": ["Goodbye!", "Bye bye!", "See you later!"],
    "default": ["I'm sorry, I don't understand.", "Could you please repeat that?", "I'm not sure I follow."],
}

# Function to generate a response
def generate_response(user_input):
    user_input = user_input.lower()
    for key in rules:
        if key in user_input:
            return random.choice(rules[key])
    return random.choice(rules["default"])

# Main function to handle the chat
def chat():
    print("Chatbot: Hi! How can I help you today?")
    while True:
```

```
        user_input = input("You: ")
        if user_input.lower() == 'exit':
            print("Chatbot: Goodbye!")
            sys.exit()

        response = generate_response(user_input)
        print("Chatbot:", response)

# Entry point of the program
if _name_ == "_main_":
    chat()
```

Prog 11:

```
import random, sys

wins, lose, tie =0 , 0 , 0
print("Enter (r)ock , (p)aper , (s)cissior or (q)uit")

turn ='p'


while turn != 'q':
    print(f"WINS: {wins}, Looses:{lose}, Tie:{tie}")
    print("Enter your choice")
    turn =input()

    if turn=='q':
        print("thank you for playing")
        sys.exit()
```

```
elif turn == 'r':
   print("Rock  verus....")
elif turn == 'p':
   print("Paper  verus....")
else:
   print("Scissor  verus....")


comp_choice=  random.randint(1,3)
# 1- Rock 2- Paper 3- Scissors
if comp_choice  == 1:
   print('ROCK')
   #Tie
   if turn =='r':
      print('Tie')
      tie= tie+1
   #lose
   elif turn =='p':
      print('You  Win!')
      wins =wins +1
   #win
   else:
      print('You  Lose :(')
      lose=lose+1

elif comp_choice  == 2:
   print('PAPER')
    #Tie
   if turn =='p':
      print('Tie')
      tie= tie+1
```

```
    #lose
    elif turn =='s':
        print('You  Win!')
        wins =wins +1
    #win
    else:
        print('You  Lose :(')
        lose=lose+1

elif comp_choice  == 3:
    print('Scissor')
     #Tie
    if turn =='s':
        print('Tie')
        tie= tie+1
    #lose
    elif turn =='r':
        print('You  Win!')
        wins =wins +1
    #win
    else:
        print('You  Lose :(')
        lose=lose+1
```