Exp: 6: Solve and Implement Water jug problem using A* Algorithm.

Program:

```
import heapq

def water_jug_a_star(jug1_capacity, jug2_capacity, target_amount):
    start_state = (0, 0)
    open_list = [(0, start_state)]  # (f-value, state)
    closed_set = set()

    while open_list:
        current_cost, current_state = heapq.heappop(open_list)
        if current_state == (target_amount, 0) or current_state == (0,
target_amount):
            # Goal reached
            return current_state

        closed_set.add(current_state)

        # Generate successor states
        successors = generate_successors(current_state, jug1_capacity,
jug2_capacity)

        if successors is not None:
            for successor in successors:
                if successor not in closed_set:
                    # Calculate f-value (cost + heuristic)
                    cost = current_cost + 1  # Assuming each step has a
cost of 1
                    heuristic = calculate_heuristic(successor,
target_amount)
                    f_value = cost + heuristic

                    heapq.heappush(open_list, (f_value, successor))

    # No solution found
    return None

def generate_successors(state, jug1_capacity, jug2_capacity):
    # Basic implementation for generating successors
    jug1, jug2 = state
    successors = []

    # Fill jug 1
    successors.append((jug1_capacity, jug2))

    # Fill jug 2
    successors.append((jug1, jug2_capacity))

    # Empty jug 1
    successors.append((0, jug2))

    # Empty jug 2
    successors.append((jug1, 0))

    # Pour water from jug 1 to jug 2
    pour = min(jug1, jug2_capacity - jug2)
    successors.append((jug1 - pour, jug2 + pour))

    # Pour water from jug 2 to jug 1
    pour = min(jug2, jug1_capacity - jug1)
    successors.append((jug1 + pour, jug2 - pour))

    return successors

def calculate_heuristic(state, target_amount):
    # Basic heuristic: Absolute difference between the total amount in
both jugs and the target amount
    return abs(sum(state) - target_amount)

# Example usage:
result = water_jug_a_star(4, 3, 2)
print(result)
```

```
(0, 2)
```

Explanation:

Import Libraries:

The code begins by importing the heapq module, which is used for managing the priority queue in the A* algorithm.

Define A Function:*

water_jug_a_star(jug1_capacity, jug2_capacity, target_amount): This function takes the capacities of two water jugs (jug1_capacity and jug2_capacity) and the target amount of water (target_amount).

The initial state is set to (0, 0), representing empty jugs.

The open_list is initialized with the start state and its associated f-value (cost + heuristic), while the closed_set keeps track of explored states.

Main Loop:

The code enters a loop that continues until the open list is empty.

In each iteration, the state with the lowest f-value is popped from the open list.

If the current state is the goal state, the function returns the goal state.

Generate Successors:

The current state is added to the closed set.

Successor states are generated using the generate_successors function, considering all possible actions (fill, empty, pour).

If successors are generated, each successor is considered. If not in the closed set, its f-value is calculated and added to the open list.

Handle No Solution:

If the open list becomes empty and no goal state is reached, the function returns None to indicate no solution was found.

Generate Successors Function:

generate_successors(state, jug1_capacity, jug2_capacity): Generates possible successor states for a given state by considering all possible actions, including filling and emptying each jug and pouring water from one jug to another.

Heuristic Function:

calculate_heuristic(state, target_amount): Provides a basic heuristic. It calculates the absolute difference between the total amount of water in both jugs and the target amount.

Example Usage:

The code includes an example usage of the water_jug_a_star function with jug capacities (4 and 3) and a target amount (2).

The result is printed to the console.