**Problem Formulation:**

**States:** People on a side of the river, boat capacity

**Initial State:** All three missionaries and all three cannibals are on different sides of the

river. The boat is on either side of the river and can only hold one or two people.

**Actions:** Move one or two people from one side to the other

**Transition Model:** Moving people updates the states

**Goal State:** All missionaries and cannibals are on the other side of the river.

**Action Cost:** One or two depending on how many people are being moved.

**Solution:** The best algorithm to use would be BFS. This algorithm guarantees an optimal solution. It is essential to avoid infinite loops, but because the state space is small, these are less likely to be an issue

This python Pseudo-code would be how to implement it

from collections import deque

def is\_valid\_state(state):

m, c = state[0], state[1]

if m < 0 or c < 0 or m > 3 or c > 3: # Check if number of missionaries or cannibals exceeds limits

return False

if m > 0 and m < c: # Cannibals outnumber missionaries

return False

if 3 - m > 0 and 3 - m < 3 - c: # Cannibals outnumber missionaries on the other side

return False

return True

def generate\_moves(state):

moves = []

m, c, b = state[0], state[1], state[2]

possible\_moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

for move in possible\_moves:

if b == 'left':

new\_state = (m - move[0], c - move[1], 'right')

else:

new\_state = (m + move[0], c + move[1], 'left')

if is\_valid\_state(new\_state):

moves.append(new\_state)

return moves

def bfs():

start\_state = (3, 3, 'left')

queue = deque([start\_state])

visited = set()

parent = {}

while queue:

current\_state = queue.popleft()

if current\_state == (0, 0, 'right'): # Goal state reached

return construct\_solution(parent, current\_state)

visited.add(current\_state)

for move in generate\_moves(current\_state):

if move not in visited:

queue.append(move)

parent[move] = current\_state

return None

def construct\_solution(parent, goal\_state):

solution = []

current\_state = goal\_state

while current\_state in parent:

solution.append(current\_state)

current\_state = parent[current\_state]

solution.append((3, 3, 'left')) # Add initial state

return solution[::-1] # Reverse the solution to get the sequence of moves

solution = bfs()

if solution:

print("Solution found:")

for i, state in enumerate(solution):

print(f"Step {i}: {state}")

else:

print("No solution found.")

**Difficulty In Solution:**

The difficulty comes from the fact that it requires careful planning to solve the problem. Although it’s simple enough to understand the rules and the problem's goal, achieving it requires careful planning and consideration of all possible moves.