In [1]:

```
# Cannibals and Missionaries Problem
# In this case, the first element of state is the number of missionaries,
#   the second element of state is the number of cannibals,
#   and the third element of state is the location of the boat (1 = wrong sid
e, 0 = correct side)
start_state = (3, 3, 1)
goal_state = (0, 0, 0)

# There's probably a more pythonic way to do this, but it works for the proble
m
def get_neighbors(curr_state):
    neighbors = []
    if curr_state[2] == 1:
        neighbors.append((curr_state[0], curr_state[1] - 1, curr_state[2] - 1
))
        neighbors.append((curr_state[0], curr_state[1] - 2, curr_state[2] - 1
))
        neighbors.append((curr_state[0] - 1, curr_state[1], curr_state[2] - 1
))
        neighbors.append((curr_state[0] - 2, curr_state[1], curr_state[2] - 1
))
        neighbors.append((curr_state[0] - 1, curr_state[1] - 1, curr_state[2]
- 1))
    elif curr_state[2] == 0:
        neighbors.append((curr_state[0], curr_state[1] + 1, curr_state[2] + 1
))
        neighbors.append((curr_state[0], curr_state[1] + 2, curr_state[2] + 1
))
        neighbors.append((curr_state[0] + 1, curr_state[1], curr_state[2] + 1
))
        neighbors.append((curr_state[0] + 2, curr_state[1], curr_state[2] + 1
))
        neighbors.append((curr_state[0] + 1, curr_state[1] + 1, curr_state[2]
+ 1))
    return neighbors

# Remove invalid states
def is_valid(state):
    # Define game-limited invalid states (such as more than 3 missionaries or
 cannibals, or less than 0)
    if state[0] > 3 or state[0] < 0 or state[1] > 3 or state[1] < 0:
        return False
        # Next define states where the cannibals eat the missionaries
    if (state[1] > state[0] > 0) or (state[1] < state[0] < 3):
        return False
    return True

# Search all possible paths using BFS
# BFS was used because the amount of turn options in the first 3 turns or so i
s large
# Since diving down the wrong set of first 3 turns would make the algorithm ta
ke much longer,
#   finding a first solution happens faster consistenly with BFS (although DFS
can get lucky sometimes)
def bfs_paths(start, goal):
    queue = [(start, [start])]
```

```
    while queue:
        curr_state, path = queue.pop(0)
        for next_state in set(get_neighbors(curr_state)) - set(path):
            if is_valid(next_state):
                if next_state == goal_state:
                    yield path + [next_state]
                else:
                    queue.append((next_state, path + [next_state]))


print("All possible paths:")
for i in list(bfs_paths(start_state, goal_state)):
    print(i)

print("Shortest:", min(list(bfs_paths(start_state, goal_state))))
```

```
All possible paths:
[(3, 3, 1), (3, 1, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1),
(0, 2, 0), (0, 3, 1), (0, 1, 0), (0, 2, 1), (0, 0, 0)]
[(3, 3, 1), (3, 1, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1),
(0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]
[(3, 3, 1), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1),
(0, 2, 0), (0, 3, 1), (0, 1, 0), (0, 2, 1), (0, 0, 0)]
[(3, 3, 1), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1),
(0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]
Shortest: [(3, 3, 1), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0),
(2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (0, 2, 1), (0, 0, 0)]
```

In [ ]: