Chris Samuel Salcedo
2022-05055
CMSC 162: Introduction to Artificial Intelligence
Laboratory Exercise 2: Missionaries and Cannibals

1. What challenges did you face while implementing the game?

Of course, enforcement of constraints such as “the boat can carry at most two people” and “missionaries must never be outnumbered by cannibals” was tricky and led me to the decision to encapsulate the game state in a class with attributes for missionaries and cannibals on each side and the boat’s position which was difficult to think of an implementation especially when you have to enforce such constraints whenever the player moves. I implemented a comprehensive validation method for non-negativity, conservation, and safety rules which honestly was very time consuming.

Additionally, apart from the careful handling of rules and state representation, what made it extra harder for me was implementing the search algorithms as well as the user-configured values for the game. In the command line implementation, one of the most ridiculously blocker I faced was implementing the visual display of “M” and “C” and aligning these were actually not easy when the numbers are changing dynamically. Also, I had an attempt to implement a GUI for the game itself, however, learning a big Python library which I have never used before itself was a challenge, and although it did not make it to the final program, it still served the purpose of challenging the mind to create logic that bridges design and aesthetics.

2. What challenges did you face while implementing the search or solution to the game?

Apart from the aforementioned complexities, the systematic methodology to explore the problem space to find a sequence of valid moves to the goal state brought some more layers to it. Coding BFS and DFS had their own difficulties, and concepts in which I had to refresh myself with. With BFS, a queue of states and visited states are constructed in order to avoid infinite loops. Although BFS is guaranteed to find the shortest path (GeeksForGeeks), it does consume a lot of memory and thus it is needed that the states are stored efficiently (which __eq__, __hash__ for the GameState class) are used and should accurately track predecessors.

DFS had its own challenges as well considering the algorithmic trade-offs, which although has lower memory overhead, it does manual cycle-checking (GeeksForGeeks).

3. Discuss how this exercise applies the steps in solving a problem in AI.

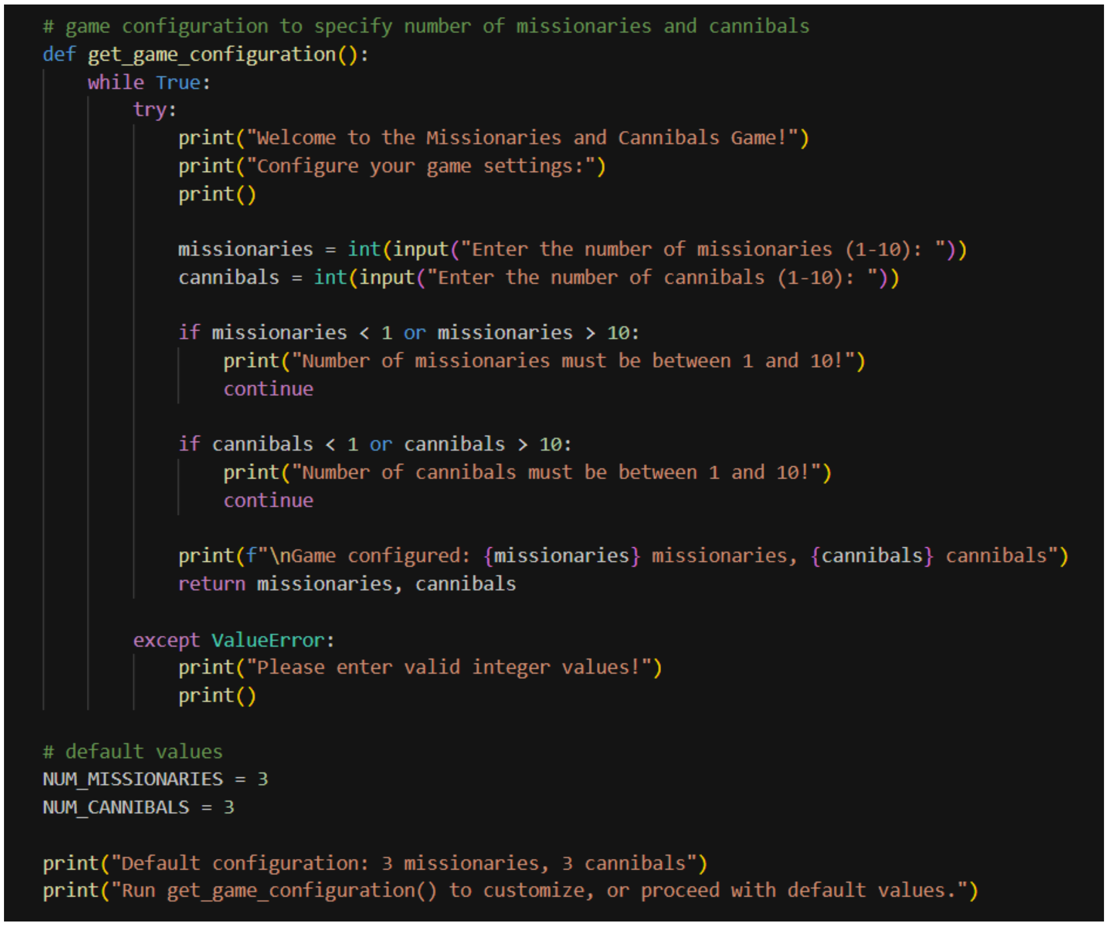
This programming exercise, substantiated by previous discussions from the lectures, mirrors the standard AI problem-solving workflow or pipeline where the problem is formally defined including the precise definition of initial states and the goal states as well as the constraints and operators associated with the problem, itself. The laboratory exercise itself required us to develop a state representation and transition model, where each configuration is represented as a GameState, similar to a point in a problem space, such that any change to one side is automatically updated and are heavily enforcing restrictions on every state.

A systemic search to find a solution path was also used such that in modern artificial intelligence, search algorithms explore the problem space to reach the goal (GeeksForGeeks). Implementation of search algorithms such as BFS and DFS, also used in AI development, illustrate exploring and pruning the search tree according to the problem's configurations.

4. For Task 2, explain the design decisions made.

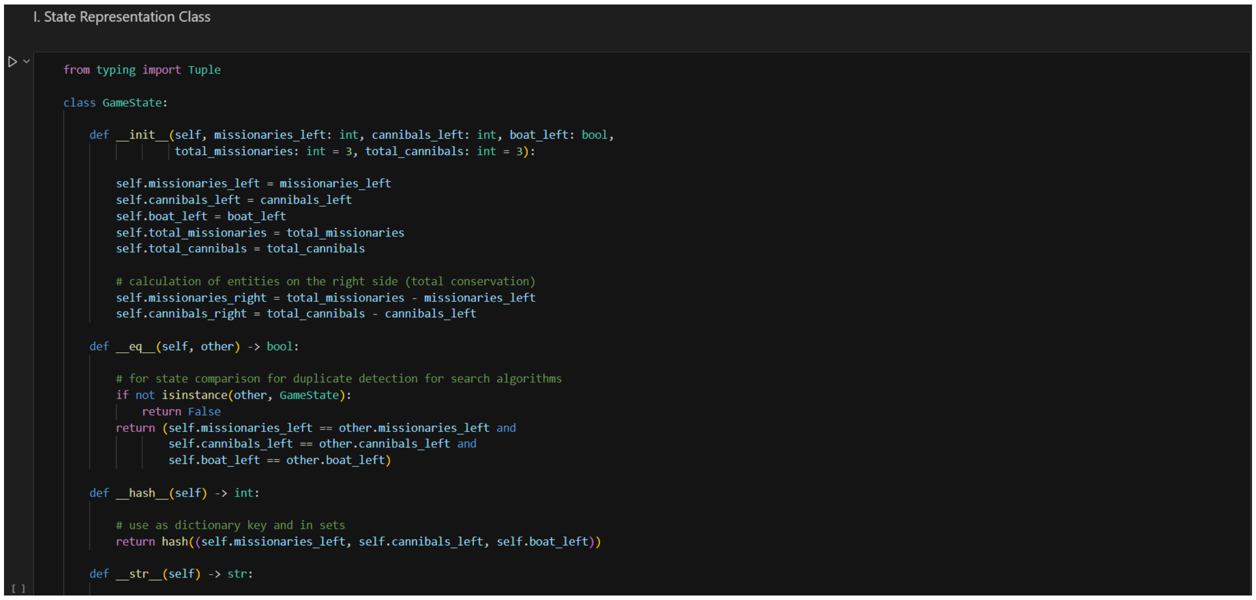
Similar to the previous laboratory exercise, an object-oriented approach is implemented which allows organization and to reflect the natural encapsulation of entities and behaviors such that the game is actually a structured system composed of states, rules, and operations and thus implementing this in OOP makes these abstractions explicit and manageable. The encapsulation of GameState class allows the grouping of critical components of a state but is more secure and error-prone rather than juggling raw tuples. In its core, it allows this state to be "self-aware", it knows how to check when it's valid, etc. Not only that OOP allowed for the modularity and separation of mechanics, interface, and search logic where each class has their own designated responsibility. Lastly, the design allowed for the code to be reusable where the functions possible_moves and apply_move methods are used whether the game is played by a human via the CLI or automatically searched via BSF/DFS.

Such design has also allowed me to implement a "customizable game" experience which allowed me to quickly alter and implement a separate class for the user's preferences for the game without heavily impacting the entire codebase.



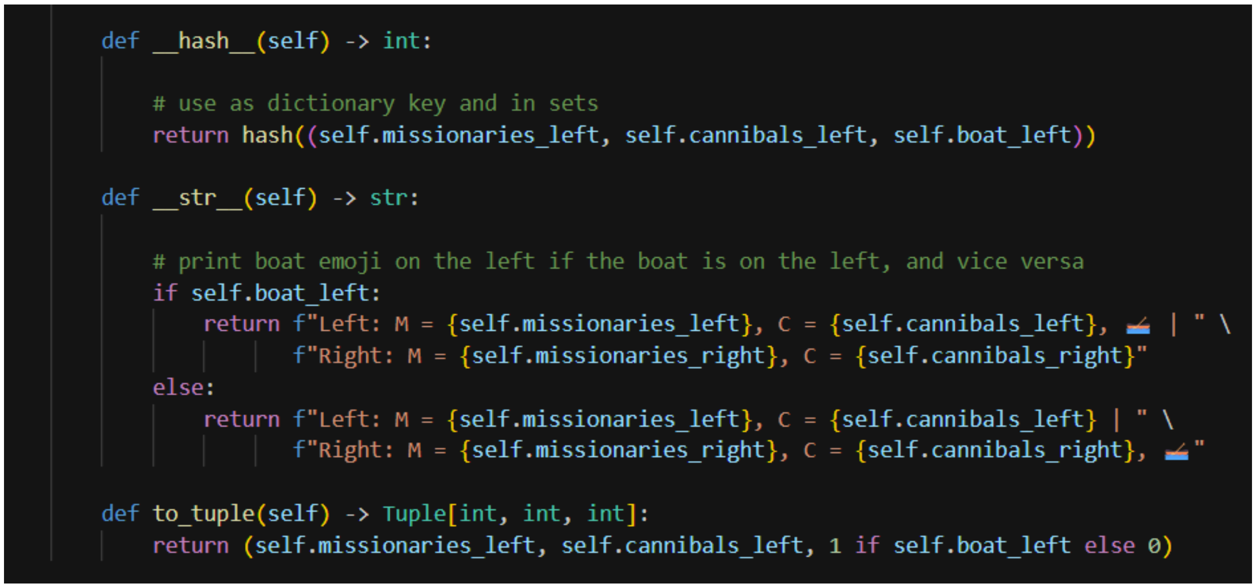
```python
# game configuration to specify number of missionaries and cannibals
def get_game_configuration():
    while True:
        try:
            print("Welcome to the Missionaries and Cannibals Game!")
            print("Configure your game settings:")
            print()

            missionaries = int(input("Enter the number of missionaries (1-10): "))
            cannibals = int(input("Enter the number of cannibals (1-10): "))

            if missionaries < 1 or missionaries > 10:
                print("Number of missionaries must be between 1 and 10!")
                continue

            if cannibals < 1 or cannibals > 10:
                print("Number of cannibals must be between 1 and 10!")
                continue

            print(f"\nGame configured: {missionaries} missionaries, {cannibals} cannibals")
            return missionaries, cannibals

        except ValueError:
            print("Please enter valid integer values!")
            print()

# default values
NUM_MISSIONARIES = 3
NUM_CANNIBALS = 3

print("Default configuration: 3 missionaries, 3 cannibals")
print("Run get_game_configuration() to customize, or proceed with default values.")
```

That being said, the program uses a configuration class that is then accessed by most preceding classes as it is one of the primary conditions of the game, for a number of missionaries and cannibals to exist in the first place.



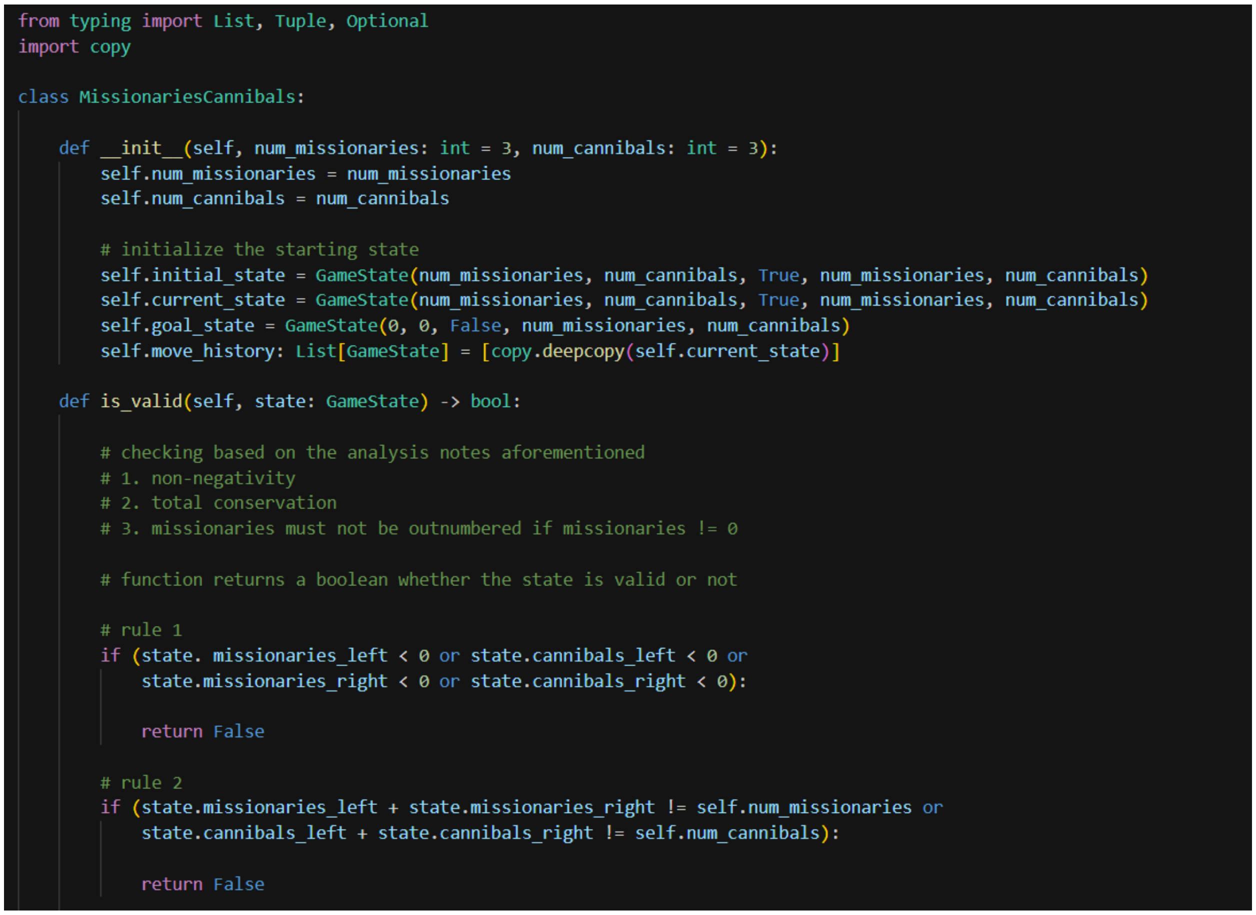
I. State Representation Class

```python
from typing import Tuple

class GameState:

    def __init__(self, missionaries_left: int, cannibals_left: int, boat_left: bool,
                 total_missionaries: int = 3, total_cannibals: int = 3):

        self.missionaries_left = missionaries_left
        self.cannibals_left = cannibals_left
        self.boat_left = boat_left
        self.total_missionaries = total_missionaries
        self.total_cannibals = total_cannibals

        # calculation of entities on the right side (total conservation)
        self.missionaries_right = total_missionaries - missionaries_left
        self.cannibals_right = total_cannibals - cannibals_left

    def __eq__(self, other) -> bool:

        # for state comparison for duplicate detection for search algorithms
        if not isinstance(other, GameState):
            return False
        return (self.missionaries_left == other.missionaries_left and
                self.cannibals_left == other.cannibals_left and
                self.boat_left == other.boat_left)

    def __hash__(self) -> int:

        # use as dictionary key and in sets
        return hash((self.missionaries_left, self.cannibals_left, self.boat_left))

    def __str__(self) -> str:
```

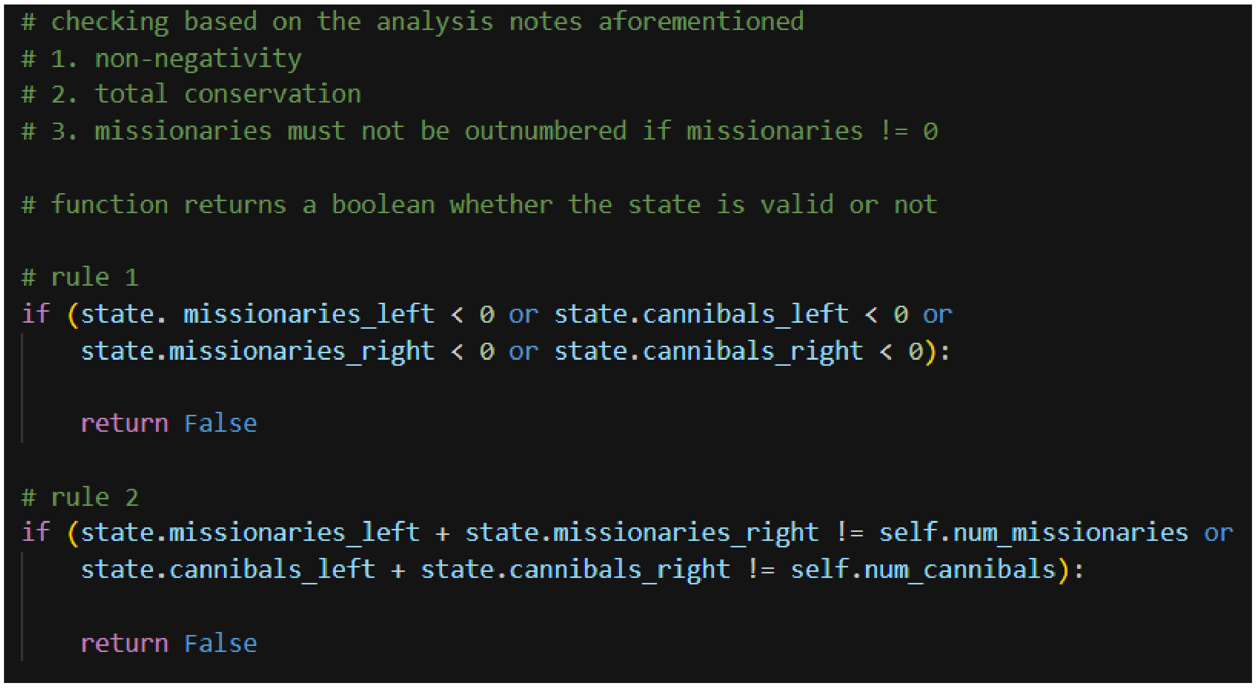


```python
    def __hash__(self) -> int:

        # use as dictionary key and in sets
        return hash((self.missionaries_left, self.cannibals_left, self.boat_left))

    def __str__(self) -> str:

        # print boat emoji on the left if the boat is on the left, and vice versa
        if self.boat_left:
            return f"Left: M = {self.missionaries_left}, C = {self.cannibals_left}, 🛶 | " \
                    f"Right: M = {self.missionaries_right}, C = {self.cannibals_right}"
        else:
            return f"Left: M = {self.missionaries_left}, C = {self.cannibals_left} | " \
                    f"Right: M = {self.missionaries_right}, C = {self.cannibals_right}, 🛶"

    def to_tuple(self) -> Tuple[int, int, int]:
        return (self.missionaries_left, self.cannibals_left, 1 if self.boat_left else 0)
```

The state representation class then follows, containing essential elements of the class that allow the storing of extra information such as total counts and computed right-bank counts as well as overriding string representation for readability. In contrast to the laboratory exercise, the boat is represented as a boolean rather than an integer as I deduced that it allowed for more semantic clarity and interpretability. The boat is on the left, true if it is but if it’s on the right then it’s false.


```python
from typing import List, Tuple, Optional
import copy

class MissionariesCannibals:

    def __init__(self, num_missionaries: int = 3, num_cannibals: int = 3):
        self.num_missionaries = num_missionaries
        self.num_cannibals = num_cannibals

        # initialize the starting state
        self.initial_state = GameState(num_missionaries, num_cannibals, True, num_missionaries, num_cannibals)
        self.current_state = GameState(num_missionaries, num_cannibals, True, num_missionaries, num_cannibals)
        self.goal_state = GameState(0, 0, False, num_missionaries, num_cannibals)
        self.move_history: List[GameState] = [copy.deepcopy(self.current_state)]

    def is_valid(self, state: GameState) -> bool:

        # checking based on the analysis notes aforementioned
        # 1. non-negativity
        # 2. total conservation
        # 3. missionaries must not be outnumbered if missionaries != 0

        # function returns a boolean whether the state is valid or not

        # rule 1
        if (state. missionaries_left < 0 or state.cannibals_left < 0 or
            state.missionaries_right < 0 or state.cannibals_right < 0):

            return False

        # rule 2
        if (state.missionaries_left + state.missionaries_right != self.num_missionaries or
            state.cannibals_left + state.cannibals_right != self.num_cannibals):

            return False
```

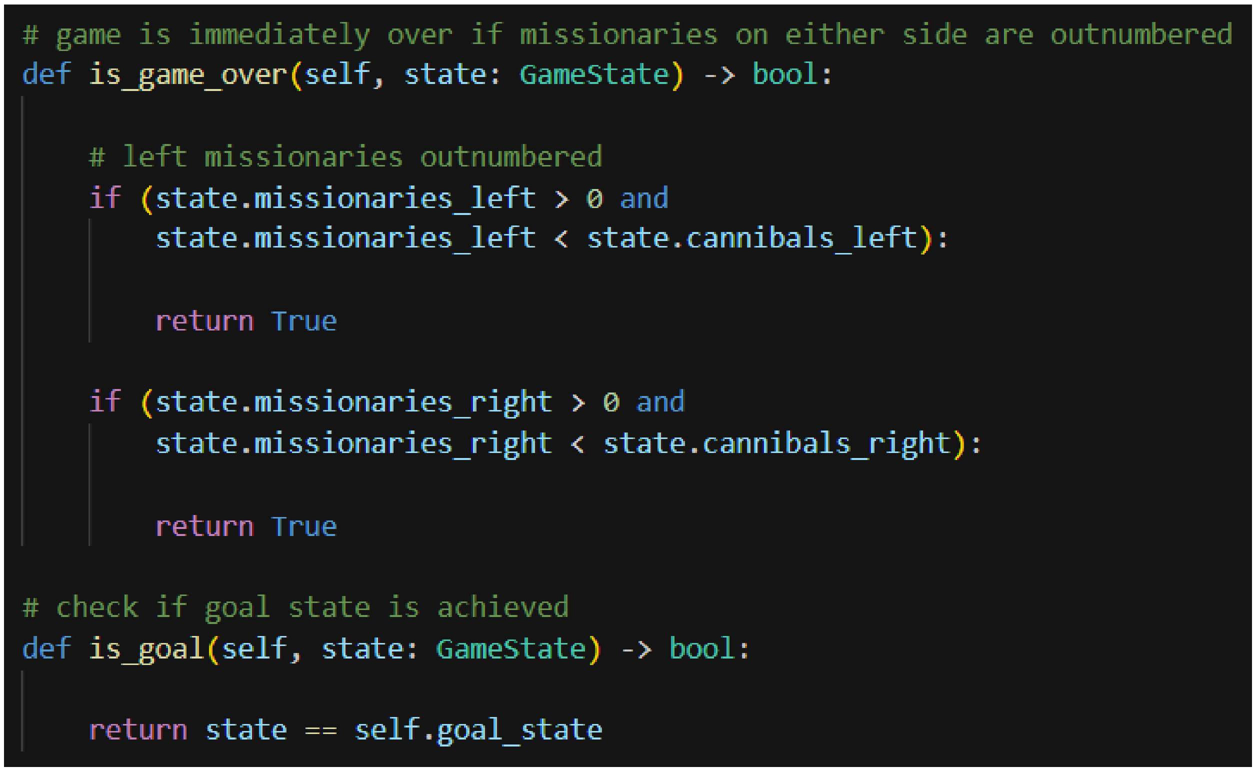
A MissionariesCannibals class is also defined, which essentially serves as the game manager or controller of the whole program. The class stores the current, which evolves as the game goes on or as the search algorithm progresses. The class also maintains a move history to record each movement for traceability and stitches the GameState tuples with the SearchAlgorithms and even the command line class. A backend service, of sorts.



```python
# checking based on the analysis notes aforementioned
# 1. non-negativity
# 2. total conservation
# 3. missionaries must not be outnumbered if missionaries != 0

# function returns a boolean whether the state is valid or not

# rule 1
if (state. missionaries_left < 0 or state.cannibals_left < 0 or
    state.missionaries_right < 0 or state.cannibals_right < 0):

    return False

# rule 2
if (state.missionaries_left + state.missionaries_right != self.num_missionaries or
    state.cannibals_left + state.cannibals_right != self.num_cannibals):

    return False
```

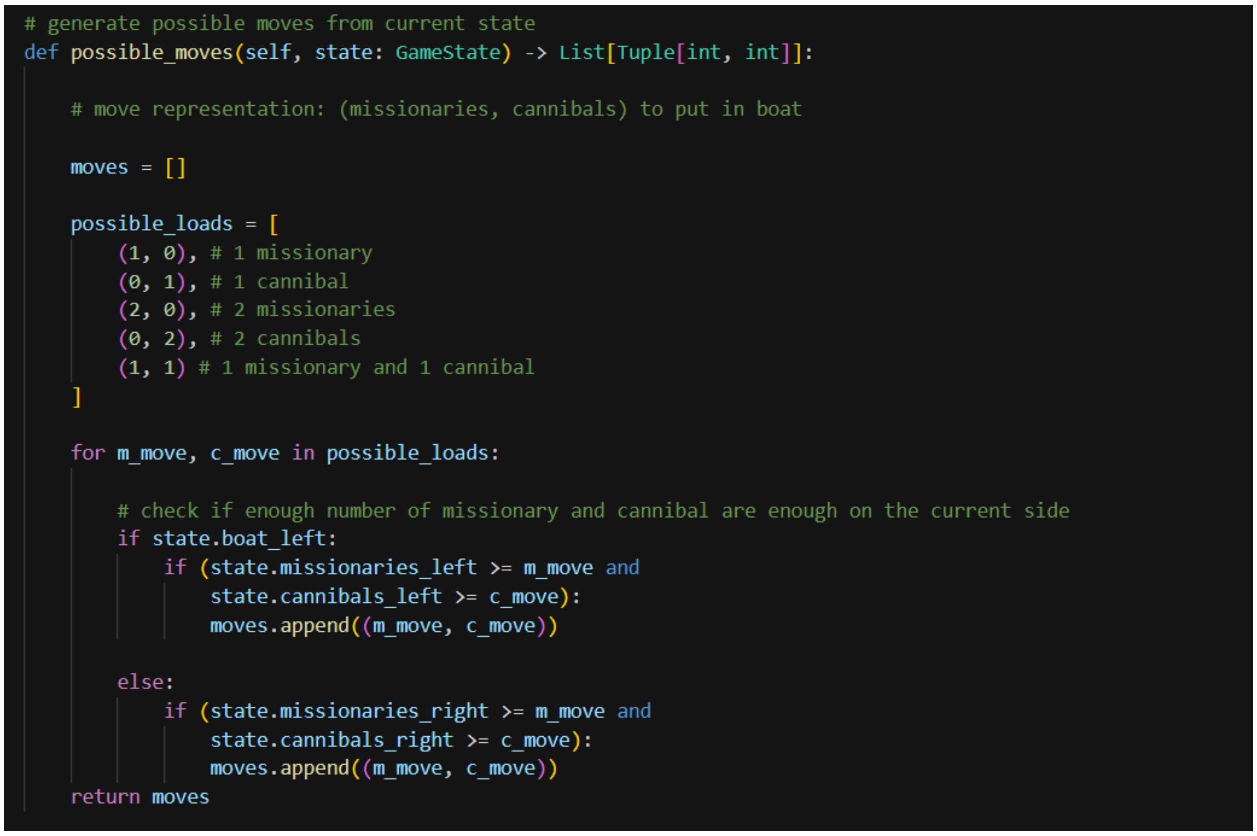
It is noted, however, that although constraint three is a legitimate rule, it is the part of the game wherein the constraint is actually allowed and mechanizes for the logic portion of the game.

Here, we defined the three rules (still in MissionariesCannibals class)



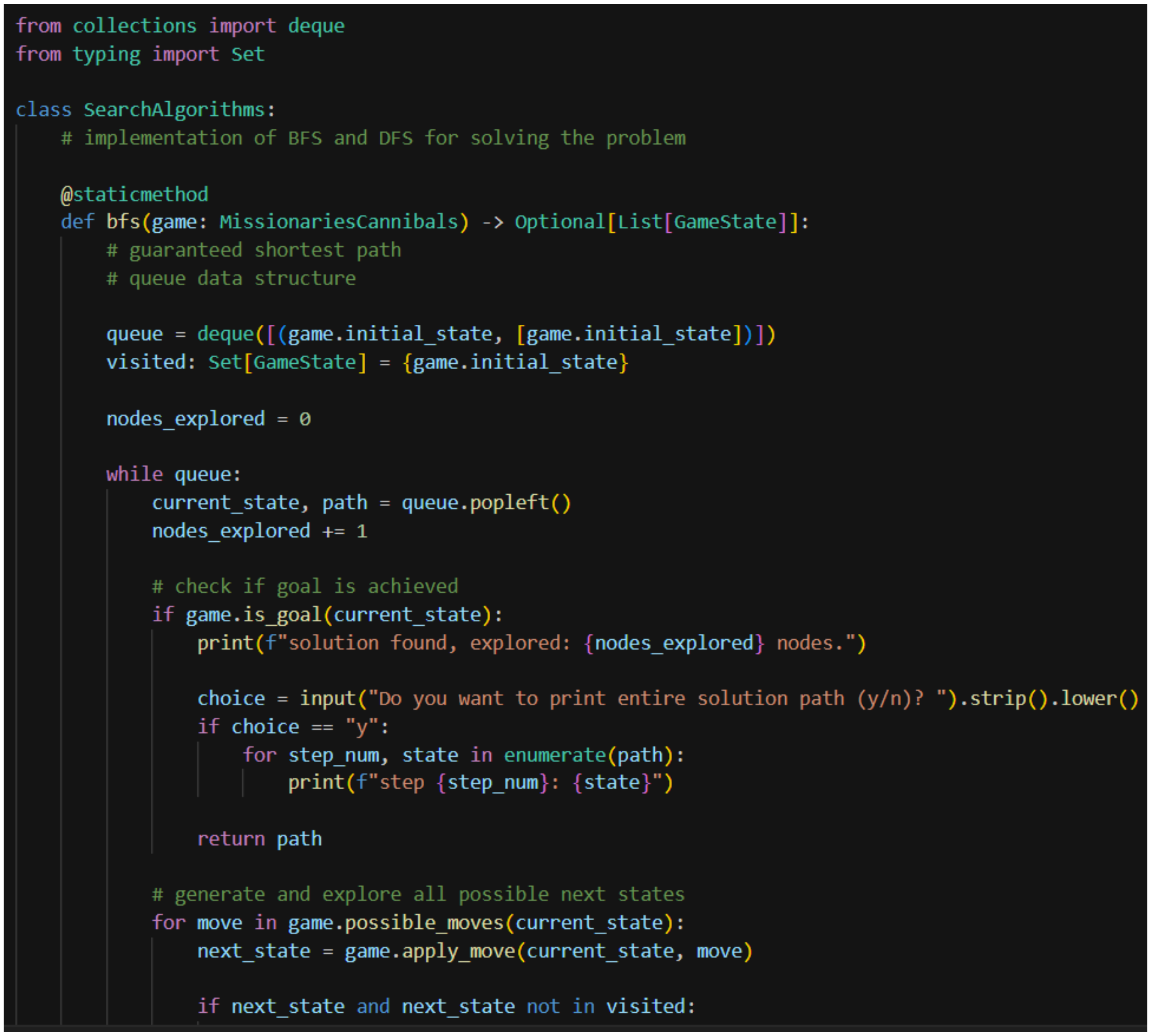
```python
# game is immediately over if missionaries on either side are outnumbered
def is_game_over(self, state: GameState) -> bool:

    # left missionaries outnumbered
    if (state.missionaries_left > 0 and
        state.missionaries_left < state.cannibals_left):

        return True

    if (state.missionaries_right > 0 and
        state.missionaries_right < state.cannibals_right):

        return True

# check if goal state is achieved
def is_goal(self, state: GameState) -> bool:

    return state == self.goal_state
```

Still in the same class, we have functions to detect whether the player has won or lost. The same class also holds functions for making moves, actually applying those moves as well as generating possible moves for the search algorithms and resetting.



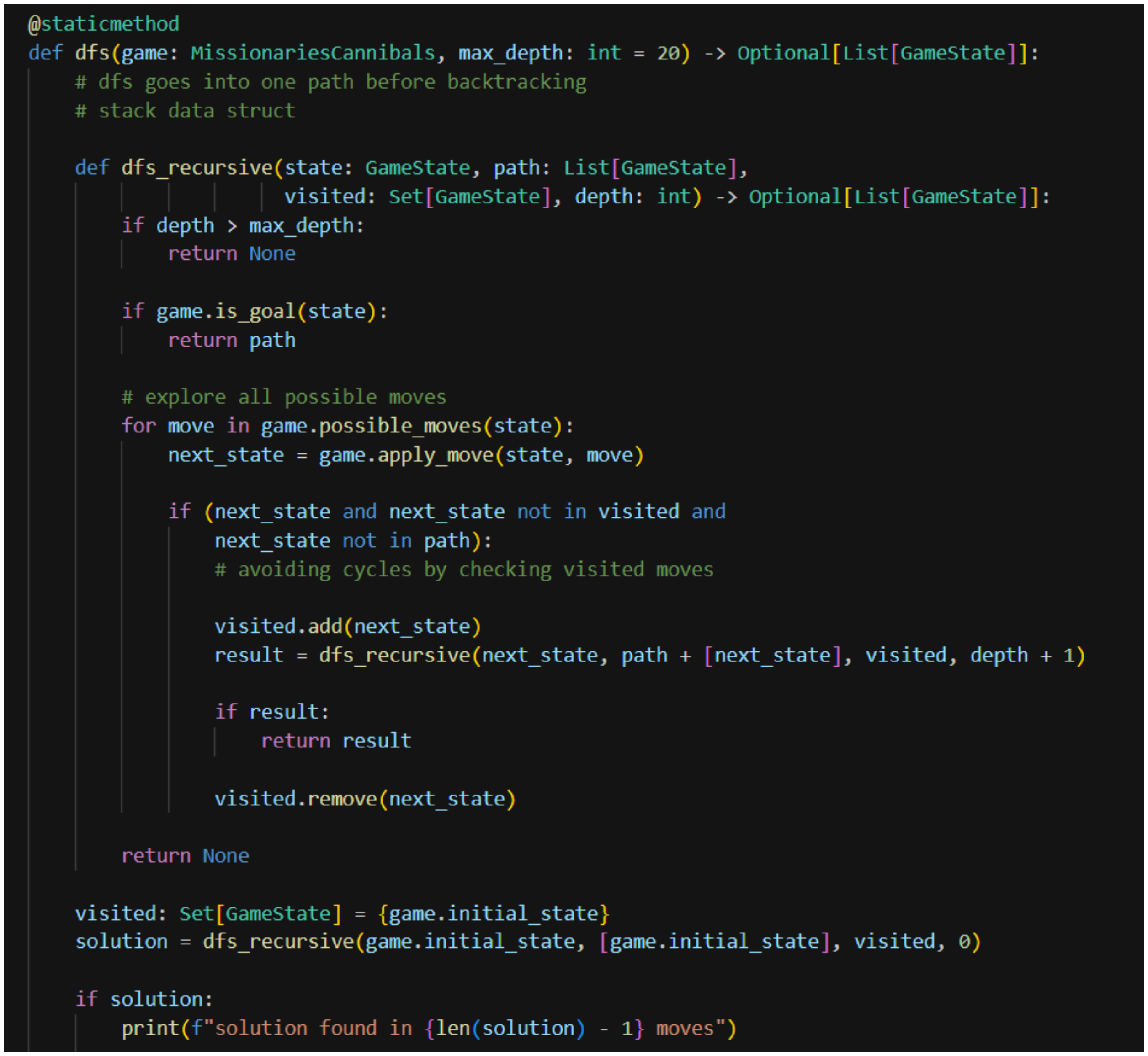
```python
# generate possible moves from current state
def possible_moves(self, state: GameState) -> List[Tuple[int, int]]:

    # move representation: (missionaries, cannibals) to put in boat

    moves = []

    possible_loads = [
        (1, 0), # 1 missionary
        (0, 1), # 1 cannibal
        (2, 0), # 2 missionaries
        (0, 2), # 2 cannibals
        (1, 1) # 1 missionary and 1 cannibal
    ]

    for m_move, c_move in possible_loads:

        # check if enough number of missionary and cannibal are enough on the current side
        if state.boat_left:
            if (state.missionaries_left >= m_move and
                state.cannibals_left >= c_move):
                moves.append((m_move, c_move))

        else:
            if (state.missionaries_right >= m_move and
                state.cannibals_right >= c_move):
                moves.append((m_move, c_move))
    return moves
```

[typo: are available on the other side]



```
from collections import deque
from typing import Set

class SearchAlgorithms:
    # implementation of BFS and DFS for solving the problem

    @staticmethod
    def bfs(game: MissionariesCannibals) -> Optional[List[GameState]]:
        # guaranteed shortest path
        # queue data structure

        queue = deque([(game.initial_state, [game.initial_state])])
        visited: Set[GameState] = {game.initial_state}

        nodes_explored = 0

        while queue:
            current_state, path = queue.popleft()
            nodes_explored += 1

            # check if goal is achieved
            if game.is_goal(current_state):
                print(f"solution found, explored: {nodes_explored} nodes.")

                choice = input("Do you want to print entire solution path (y/n)? ").strip().lower()
                if choice == "y":
                    for step_num, state in enumerate(path):
                        print(f"step {step_num}: {state}")

                return path

            # generate and explore all possible next states
            for move in game.possible_moves(current_state):
                next_state = game.apply_move(current_state, move)

                if next_state and next_state not in visited:
```

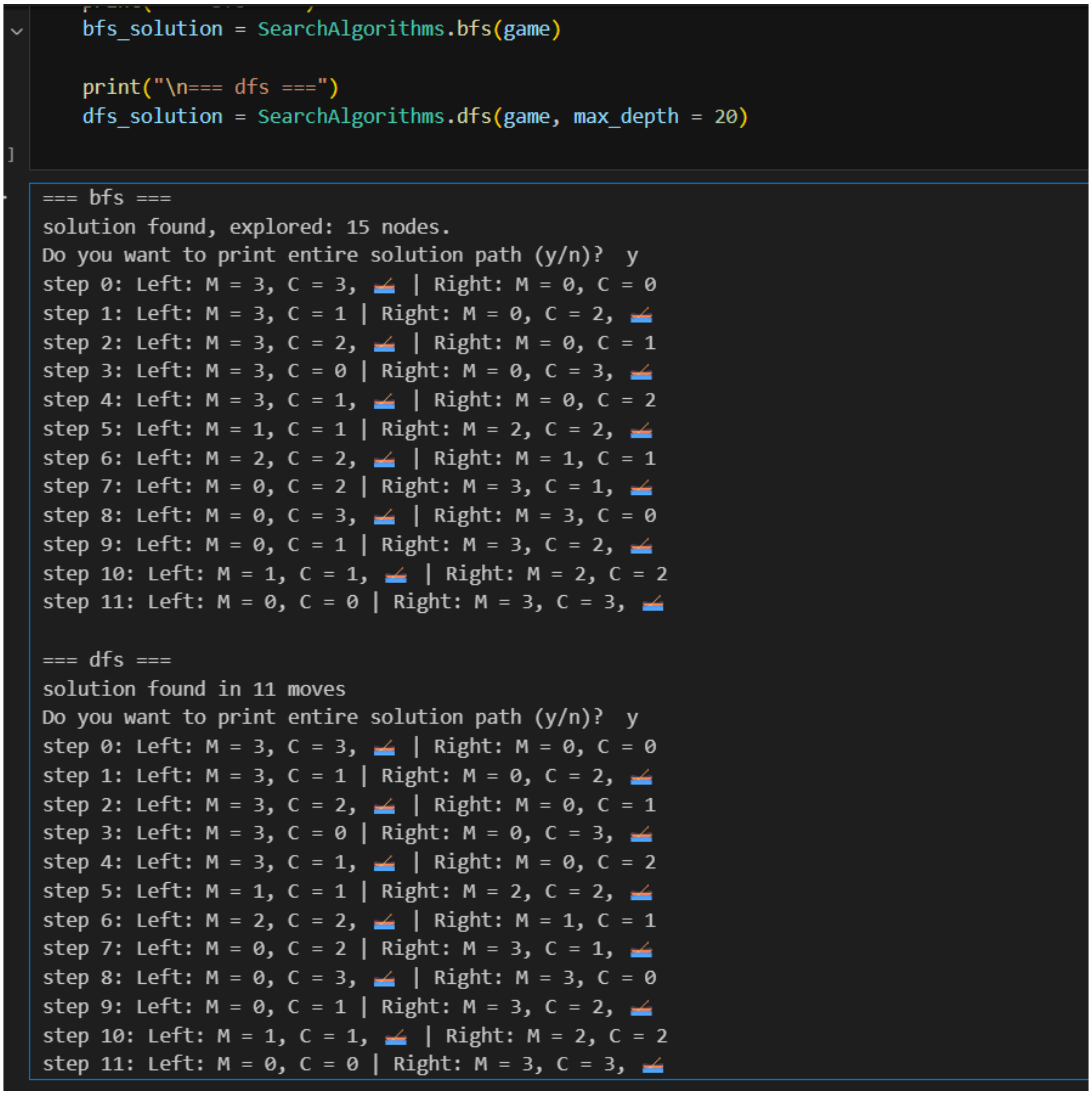
The SearchAlgorithms class holds definitions for the search algorithms. BFS utilizes a queue to systematically explore states level by level, which can be (partially seen) in the screenshot. The algorithm then guarantees to find the shortest solution as it, again, operates by level. All possible moves, calling the function from the previously discussed class, and pushing it to the queue.



```python
@staticmethod
def dfs(game: MissionariesCannibals, max_depth: int = 20) -> Optional[List[GameState]]:
    # dfs goes into one path before backtracking
    # stack data struct

    def dfs_recursive(state: GameState, path: List[GameState],
                      visited: Set[GameState], depth: int) -> Optional[List[GameState]]:
        if depth > max_depth:
            return None

        if game.is_goal(state):
            return path

        # explore all possible moves
        for move in game.possible_moves(state):
            next_state = game.apply_move(state, move)

            if (next_state and next_state not in visited and
                next_state not in path):
                # avoiding cycles by checking visited moves

                visited.add(next_state)
                result = dfs_recursive(next_state, path + [next_state], visited, depth + 1)

                if result:
                    return result

                visited.remove(next_state)

        return None

    visited: Set[GameState] = {game.initial_state}
    solution = dfs_recursive(game.initial_state, [game.initial_state], visited, 0)

    if solution:
        print(f"solution found in {len(solution) - 1} moves")
```

DSF, on the other hand, uses recursiveness and a configurable depth limit to avoid potential infinite loops aside from the implementation wherein visited states are no longer visited.

Here is the output when the SearchAlgorithms functions are called.

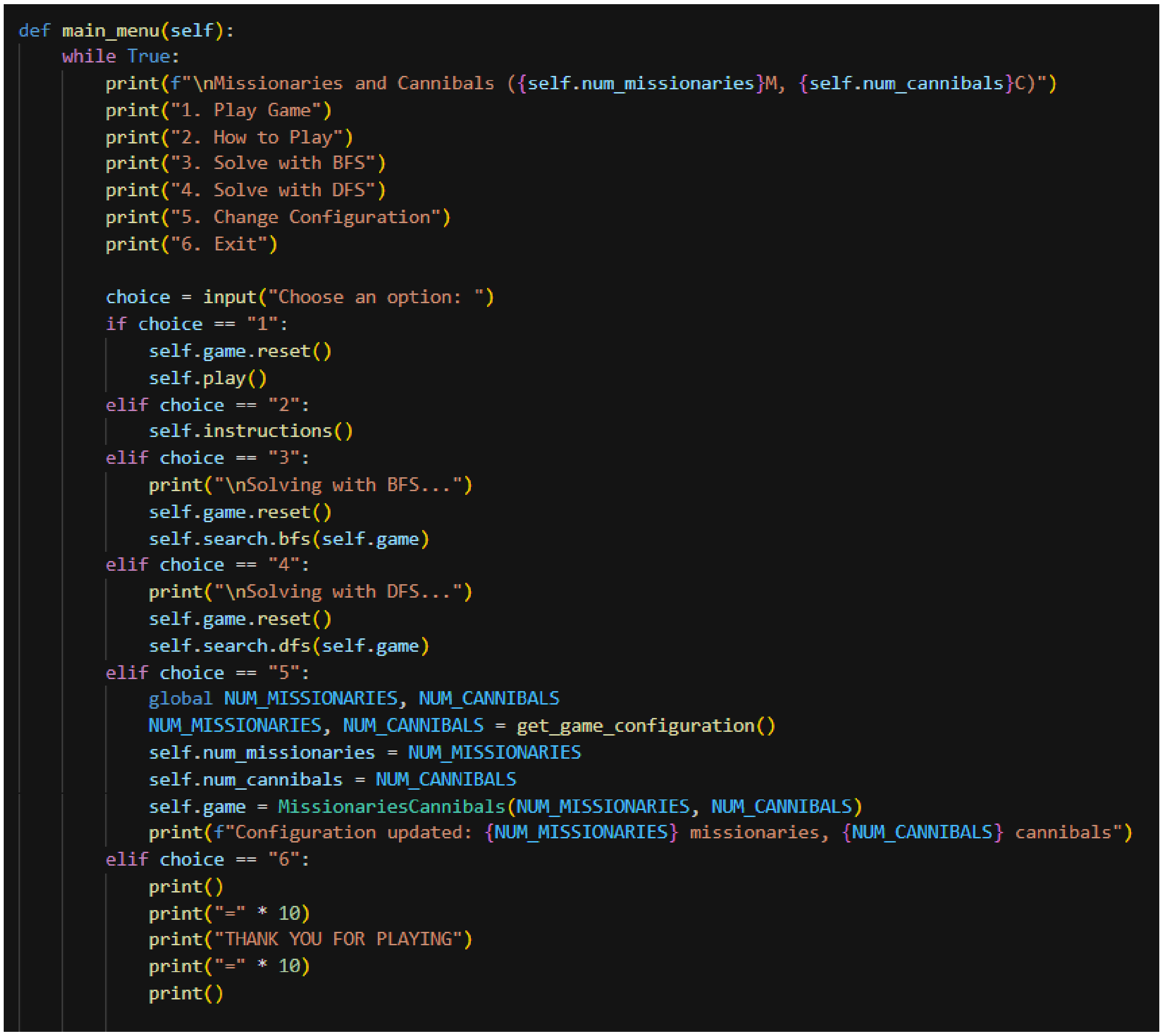


```
    bfs_solution = SearchAlgorithms.bfs(game)

    print("\n=== dfs ===")
    dfs_solution = SearchAlgorithms.dfs(game, max_depth = 20)
```

```
=== bfs ===
solution found, explored: 15 nodes.
Do you want to print entire solution path (y/n)?  y
step 0: Left: M = 3, C = 3, 🛶 | Right: M = 0, C = 0
step 1: Left: M = 3, C = 1 | Right: M = 0, C = 2, 🛶
step 2: Left: M = 3, C = 2, 🛶 | Right: M = 0, C = 1
step 3: Left: M = 3, C = 0 | Right: M = 0, C = 3, 🛶
step 4: Left: M = 3, C = 1, 🛶 | Right: M = 0, C = 2
step 5: Left: M = 1, C = 1 | Right: M = 2, C = 2, 🛶
step 6: Left: M = 2, C = 2, 🛶 | Right: M = 1, C = 1
step 7: Left: M = 0, C = 2 | Right: M = 3, C = 1, 🛶
step 8: Left: M = 0, C = 3, 🛶 | Right: M = 3, C = 0
step 9: Left: M = 0, C = 1 | Right: M = 3, C = 2, 🛶
step 10: Left: M = 1, C = 1, 🛶 | Right: M = 2, C = 2
step 11: Left: M = 0, C = 0 | Right: M = 3, C = 3, 🛶

=== dfs ===
solution found in 11 moves
Do you want to print entire solution path (y/n)?  y
step 0: Left: M = 3, C = 3, 🛶 | Right: M = 0, C = 0
step 1: Left: M = 3, C = 1 | Right: M = 0, C = 2, 🛶
step 2: Left: M = 3, C = 2, 🛶 | Right: M = 0, C = 1
step 3: Left: M = 3, C = 0 | Right: M = 0, C = 3, 🛶
step 4: Left: M = 3, C = 1, 🛶 | Right: M = 0, C = 2
step 5: Left: M = 1, C = 1 | Right: M = 2, C = 2, 🛶
step 6: Left: M = 2, C = 2, 🛶 | Right: M = 1, C = 1
step 7: Left: M = 0, C = 2 | Right: M = 3, C = 1, 🛶
step 8: Left: M = 0, C = 3, 🛶 | Right: M = 3, C = 0
step 9: Left: M = 0, C = 1 | Right: M = 3, C = 2, 🛶
step 10: Left: M = 1, C = 1, 🛶 | Right: M = 2, C = 2
step 11: Left: M = 0, C = 0 | Right: M = 3, C = 3, 🛶
```

In the DFS, the limited depth and backtracking allowed it to discover the same optimal-length path.

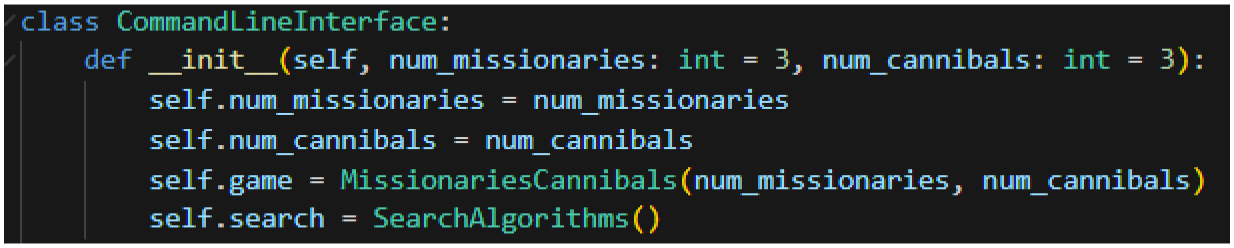
Lastly, for the CommandLineInterface class is essentially just a culmination of every other class that have been defined before utilizing GameState class tuples through MissionariesCannibals methods with implementations from the user configuration function as well everything in a what seemed like just a simple menu.



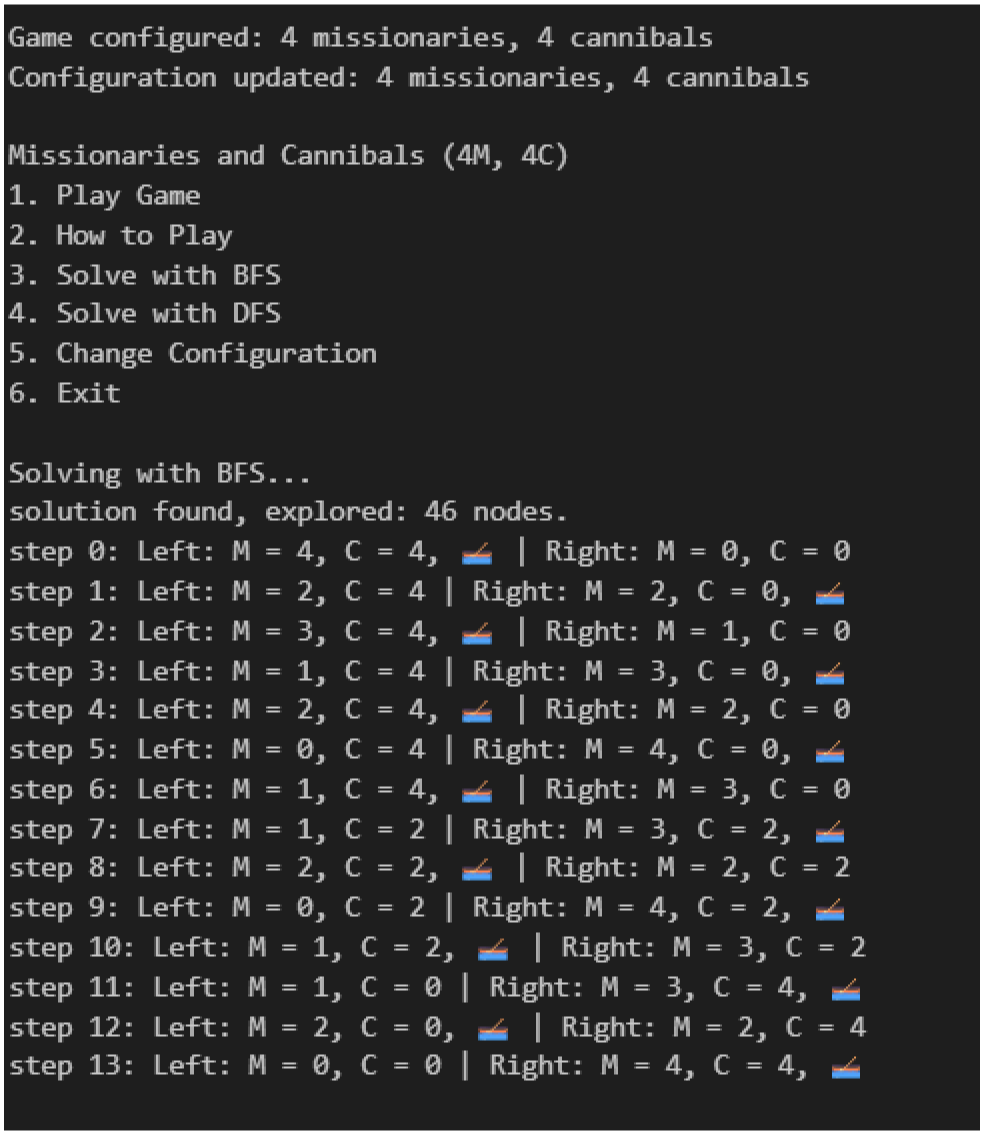
```python
def main_menu(self):
    while True:
        print(f"\nMissionaries and Cannibals ({self.num_missionaries}M, {self.num_cannibals}C)")
        print("1. Play Game")
        print("2. How to Play")
        print("3. Solve with BFS")
        print("4. Solve with DFS")
        print("5. Change Configuration")
        print("6. Exit")

        choice = input("Choose an option: ")
        if choice == "1":
            self.game.reset()
            self.play()
        elif choice == "2":
            self.instructions()
        elif choice == "3":
            print("\nSolving with BFS...")
            self.game.reset()
            self.search.bfs(self.game)
        elif choice == "4":
            print("\nSolving with DFS...")
            self.game.reset()
            self.search.dfs(self.game)
        elif choice == "5":
            global NUM_MISSIONARIES, NUM_CANNIBALS
            NUM_MISSIONARIES, NUM_CANNIBALS = get_game_configuration()
            self.num_missionaries = NUM_MISSIONARIES
            self.num_cannibals = NUM_CANNIBALS
            self.game = MissionariesCannibals(NUM_MISSIONARIES, NUM_CANNIBALS)
            print(f"Configuration updated: {NUM_MISSIONARIES} missionaries, {NUM_CANNIBALS} cannibals")
        elif choice == "6":
            print()
            print("=" * 10)
            print("THANK YOU FOR PLAYING")
            print("=" * 10)
            print()
```

Notably, option 5 calls for the configuration function, and of course, a default value of 3 has also been implemented in case the user has no intention of changing that.

Here's the class initiating objects from the previous classes,



```python
class CommandLineInterface:
    def __init__(self, num_missionaries: int = 3, num_cannibals: int = 3):
        self.num_missionaries = num_missionaries
        self.num_cannibals = num_cannibals
        self.game = MissionariesCannibals(num_missionaries, num_cannibals)
        self.search = SearchAlgorithms()
```



```
Game configured: 4 missionaries, 4 cannibals
Configuration updated: 4 missionaries, 4 cannibals

Missionaries and Cannibals (4M, 4C)
1. Play Game
2. How to Play
3. Solve with BFS
4. Solve with DFS
5. Change Configuration
6. Exit

Solving with BFS...
solution found, explored: 46 nodes.
step 0: Left: M = 4, C = 4, 🛶 | Right: M = 0, C = 0
step 1: Left: M = 2, C = 4 | Right: M = 2, C = 0, 🛶
step 2: Left: M = 3, C = 4, 🛶 | Right: M = 1, C = 0
step 3: Left: M = 1, C = 4 | Right: M = 3, C = 0, 🛶
step 4: Left: M = 2, C = 4, 🛶 | Right: M = 2, C = 0
step 5: Left: M = 0, C = 4 | Right: M = 4, C = 0, 🛶
step 6: Left: M = 1, C = 4, 🛶 | Right: M = 3, C = 0
step 7: Left: M = 1, C = 2 | Right: M = 3, C = 2, 🛶
step 8: Left: M = 2, C = 2, 🛶 | Right: M = 2, C = 2
step 9: Left: M = 0, C = 2 | Right: M = 4, C = 2, 🛶
step 10: Left: M = 1, C = 2, 🛶 | Right: M = 3, C = 2
step 11: Left: M = 1, C = 0 | Right: M = 3, C = 4, 🛶
step 12: Left: M = 2, C = 0, 🛶 | Right: M = 2, C = 4
step 13: Left: M = 0, C = 0 | Right: M = 4, C = 4, 🛶
```

Finally, here's a sample output program where I configured it to 4 M and 4 C and used BFS to search for a solution.