LAB02 Report:

Detect Error :

- In generate_full_space_tree.py, line 125, can't find i . Solve : delete the command line of line 101

- In solve.py , line 145 , in show_solution
  ```
  if node.get_label() != str(self.start_state):
     ^^^^^^^^^^^^^^
  ```
  AttributeError: 'list' object has no attribute 'get_label'
  solve : the code trying to call get_label() on a list object. The error occurs because nodes contains list objects instead of graph nodes as expected.
  Let's fix the show_solution method. Fix :
  Removed the nodes list since it wasn't being used correctly
  Removed the node styling code since it was causing the error
  Simplified the move tracking and state representation
  Fixed string formatting for better readability
  Added reverse() to show path from start to goal instead of goal to start
  Added safety checks for Move and Parent dictionary lookups
- Change library use from pydot to pydot_ng (because my pydot can't draw legend)

  Origin :

```
def show_solution(self):
        # Recursively start from Goal State
        # And find parent until start state is reached

        state = self.goal_state
        path, steps, nodes = [], [], []

        while state is not None:
            path.append(state)
            steps.append(Move[state])
            nodes.append(node_list[state])

            state = Parent[state]

        steps, nodes = steps[:-1], nodes[:-1]

        number_missionaries_left, number_cannibals_left = 3, 3
        number_missionaries_right, number_cannibals_right = 0, 0

        print("-" * 60)
```

```
        self.draw(number_missionaries_left=number_missionaries_left,
number_cannibals_left=number_cannibals_left,
                number_missionaries_right=number_missionaries_right,
number_cannibals_right=number_cannibals_right)
        for i, ((number_missionaries, number_cannibals, side), node) in
enumerate(zip(steps[1:], nodes[1:])):
                if node.get_label() != str(self.start_state):
                    node.set_style("filled")
                    node.set_fillcolor("yellow")

                print(f"Step {i + 1}: Move {number_missionaries} missionaries and
{number_cannibals} \
                    cannibals from {self.boat_side[side]} to {self.boat_side[int(not side)]}.")

                op = -1 if side == 1 else 1

                number_missionaries_left = number_missionaries_left + op *
number_missionaries
                number_cannibals_left = number_cannibals_left + op * number_cannibals

                number_missionaries_right = number_missionaries_right - op *
number_missionaries
                number_cannibals_right = number_cannibals_right - op * number_cannibals

                self.draw(number_missionaries_left=number_missionaries_left,
number_cannibals_left=number_cannibals_left,
                        number_missionaries_right=number_missionaries_right,
number_cannibals_right=number_cannibals_right)
        print("Congratulations!!! you have solved the problem")
        print("*" * 60)
```

Fix :

```
    def show_solution(self):
    # Recursively start from Goal State
    # And find parent until start state is reached

    state = self.goal_state
    path, steps = [], []

    while state is not None:
```

```python
        path.append(state)
        steps.append(Move[state]) if state in Move else steps.append(None)
        state = Parent[state] if state in Parent else None

    steps = steps[:-1]  # Remove the last None move
    path.reverse()  # Get path from start to goal
    steps.reverse()  # Get moves from start to goal

    number_missionaries_left, number_cannibals_left = 3, 3
    number_missionaries_right, number_cannibals_right = 0, 0

    print("-" * 60)
    self.draw(number_missionaries_left=number_missionaries_left,
              number_cannibals_left=number_cannibals_left,
              number_missionaries_right=number_missionaries_right,
              number_cannibals_right=number_cannibals_right)

    for i, (number_missionaries, number_cannibals, side) in enumerate(steps):
        print(f"Step {i + 1}: Move {number_missionaries} missionaries and
{number_cannibals} "
              f"cannibals from {self.boat_side[side]} to {self.boat_side[int(not side)]}.")

        op = -1 if side == 1 else 1

        number_missionaries_left += op * number_missionaries
        number_cannibals_left += op * number_cannibals

        number_missionaries_right -= op * number_missionaries
        number_cannibals_right -= op * number_cannibals

        self.draw(number_missionaries_left=number_missionaries_left,
                  number_cannibals_left=number_cannibals_left,
                  number_missionaries_right=number_missionaries_right,
                  number_cannibals_right=number_cannibals_right)

    print("Congratulations!!! You have solved the problem")
    print("*" * 60)
```

Explain source code after fix

1.Generate_full_space_tree.py

After fixing, the generate_full_space_tree.py generates a visual representation (state space tree) of the classic "Missionaries and Cannibals" puzzle. Let me break down what it does:

1. The Puzzle Context:
- The "Missionaries and Cannibals" puzzle involves moving 3 missionaries and 3 cannibals across a river using a boat
- The goal is to move everyone to the other side
- Key rule: Cannibals can never outnumber missionaries on either side (or the missionaries are in danger)
2. Key Functionality:
- The code creates a visual graph showing all possible states and moves in the puzzle
- Each node represents a state with format (missionaries, cannibals, side)
- Side is represented as 1 (starting side) or 0 (destination side)
- The graph is color-coded:
    - Orange: Valid intermediate states
    - Red: Invalid states (where cannibals outnumber missionaries)
    - Green: Goal state (everyone reached other side)
    - Grey: Dead-end states
    - White: Starting state (3,3,1)
3. Implementation Details:
- Uses `pydot_ng` to generate the graph visualization
- Implements breadth-first search (using `deque`) to explore all possible states
- Allows setting a maximum depth for the tree via command line argument
- Generates a PNG file showing the complete state space tree
- Tracks parent states to build the tree structure
- Validates moves using rules like:
    - Can't have negative people
    - Can't have more than 3 people on either side
    - Cannibals can't outnumber missionaries
4. Valid Moves:
- The `options` list defines possible boat moves: `[(1, 0), (0, 1), (1, 1), (0, 2), (2, 0)]`
    - First number: missionaries to move
    - Second number: cannibals to move
    - For example, (1,1) means move 1 missionary and 1 cannibal

To use this code, you would:

1. Ensure Graphviz is installed (it's required for visualization)
2. Run the script, optionally specifying a depth limit: `python generate_full_space_tree.py --depth 10`
3. Get a PNG file showing all possible states and transitions in the puzzle

2.Solve.py

This code implements a solution to the classic "Missionaries and Cannibals" puzzle using both Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms

The code solves the following puzzle:

1. Problem Statement:
   - There are 3 missionaries and 3 cannibals on one side of a river
   - They need to cross to the other side using a boat
   - The boat can carry at most 2 people
   - At no point can cannibals outnumber missionaries on either side (or the cannibals will eat the missionaries)
2. Key Features:
   - Uses both DFS and BFS search algorithms to find a solution
   - Creates a visual state space tree using pydot_ng (GraphViz)
   - Displays the solution steps using emoji visualizations
   - Tracks the path to solution using Parent and Move dictionaries
3. State Representation:
   - Each state is represented as a tuple (m, c, s) where:
     - m: number of missionaries on the left side
     - c: number of cannibals on the left side
     - s: boat side (1 for left, 0 for right)
   - Start state: (3, 3, 1)
   - Goal state: (0, 0, 0)
4. Available Moves:
   - The possible moves are: (1,0), (0,1), (1,1), (0,2), (2,0)
   - Each tuple represents (missionaries, cannibals) to move
5. Visualization Features:
   - Creates a colored graph showing the search space
   - Blue: Start node
   - Green: Goal node
   - Red: Invalid states (cannibals eat missionaries)
   - Yellow: Solution path
   - Gray: Dead ends
   - Orange: Expandable nodes
6. Solution Display:
   - Shows step-by-step solution with emoji representations
   - Displays missionaries as 👴 and cannibals as 👹
   - Shows the river crossing process with a visual representation

To use this code, you would:

1. Initialize the Solution class
2. Call the solve method with either "dfs" or "bfs"
3. Use show_solution() to see the step-by-step solution
4. Use write_image() to save the state space tree visualization

3. Main.py

This is the main driver code that runs the Missionaries and Cannibals puzzle solver. Let me explain how it works:

1. Command Line Arguments:
   - -m or --method: Specifies which search method to use (BFS or DFS)
   - -l or --legend: Determines whether to include a legend in the output graph
2. Usage Examples:

```
# Run with BFS (default) and no legend
python main.py

# Run with DFS (default) and no legend
python main.py -m dfs
# Run with DFS and include legend
python main.py -m dfs -l True

# Run with BFS and include legend
python main.py --method bfs --legend True
```

Program Flow:

- Creates a Solution instance
- Calls solve() with specified method (defaults to "bfs")
- If solution is found:
  - Shows step-by-step solution on console
  - Generates output file name based on method and legend flag
  - Optionally adds legend to visualization
  - Saves state space tree as PNG file
- If no solution found, raises an exception

Output Files:

- Without legend: bfs.png or dfs.png
- With legend: bfs_legend.png or dfs_legend.png