Name: WANG, Tsunying Student no.: 18078956d Program: Broad Discipline of Computing

[Question Description & Data Abstraction]

The goal of this problem is to send three couples in green, blue and red to cross a river using a boat, from east to west. The boat needs at least one person to drive and can fit at most two people in one ride. In addition, during the process, a husband cannot let his wife (same color) be with another man without his presence.

In this project, I will solve this problem by showing step-by-step approach using change of states after each move is made. In each state, there will be a letter in lower case and 6 letters in upper case, for example "eEWEWEE". The lower case represents the boat's position, while the other 6 represents the three couples in the order of: green wife, green husband, blue wife, blue husband, red wife, and red husband. In general, an individual letter can either be 'e' representing character on the east side, otherwise 'w' representing character on the west.

As a result of data abstraction, there will be 128 illegal states, having 42 states being legal. Moreover, after implementation of the graph, it was found that the graph can generate different shortest path. However, it was observed that they share a common trait by meeting at 'wWEWEWE' and 'eEWEWEW' during their 3rd move and 8th move. For details of the modular design and algorithm behind, I will be explaining on the rest of the report.

[Python Modular Design & Algorithm Explanation for graph]

For the Python implementation, I have used a string to represent a state, a list for storing both legal and illegal states, a dictionary for graph, various sets for values of the dictionary, a list for shortest path, and 11 strings the for final output. These are structured by implementing 6 functions, which are: slover, genState, genGraph, isStateLegal, isMoveLegal, and genTrip. I have also used a function of findShortestPath which was obtained from outside source. The isStateLegal and isMovelegal are sub-functions which are included under the genGraph function. Each function has its own functionalities and purpose of implementation.

The Solver function is the main body of the program which designed to call the functions of genState, genGraph, findShortestPath, and genTrip. The functions are stored into different variables so that interactions between the functions are possible. The function call of the solver is located at the end of the code.

The genState function is designed to output a list containing strings of all illegal states. The function consists of 7 for loops, one being nested under the previous one, iterating the letter of "E" and "W" (position of east and west). At the end of the last loop, the function will add all the letter into a single string and append to the local list of allState, the output of this function. As there are 7 loops consists of two iteration, in total there will be number of 2^7 or 128 illegal states.

The genGraph function is designed to input the illegal list, output a dictionary containing keys of legal states, and its respective neighbor states in a set as its values. The function will first run a for loop, iterating each item in the list of illegal states, and run the isStateLegal function to check whether the function is legal or not. If the state found to be legal, add to the legal list. Then by using the legal list, will use two for loops, one loop for picking a state in the list, and another one is to check all of the states to see which of them are neighbors of the selected state. This comparison is done by use of isMoveLegal function and will add to the graph with as soon as one iteration of the outer loop ends. When the graph settles, return the graph as the output.

To mention about the algorithm behind the isStateLegal. The function is designed to check the main rule of the question: avoiding a wife being with other husbands without her husband's presence. This function will therefore input a string as the state and return a Boolean data type: True if state is legal, False for illegal. The function will first run three similar if functions, for three different couples. Each if statements check whether the couples are in the opposite sites, and check whether the wife is with

other man or not. If the state meets any of these rules will then return false. In addition, there is another elif function, to check whether the state is specific strings of ‘eWWWWWW’ or ‘wEEEEEE’, if the condition is true, return False. This is written to exclude the illegal cases those are impossible to happen: boat is located on the another side during the start, and finish with boat not arriving to west. If none of all four if statement is satisfied, then the function will return Ture in a else statement. At the end, it was found there are total of 42 legal states out of 128.

The isMoveLegal compares two states as inputs to check whether they are neighbors or not. If they are connected, the function will return True as Boolean datatype, otherwise return False. This entire function is structured in three if statements as filters, if any of these if conditions are not satisfied, will return False value. The first filter is to check whether the boat had move or not by checking the first letters of two states, they must be different so that it means the boat had moved. The second filter is made of one for statement and two if statements. The for statements with the first statement triggers the condition of checking the each character whether there are differences, and another if statement is to check the move is moveable link, (the person moving has a boat to move), return False if the difference is on a not moveable person. Lastly, the third filter is to check if the boat was driven at least a person and at most two people which is the rule of the problem.

As soon as the genGraph ends, the graph must have completed. By observation, it is found that there are two legal states are not useful because there is no valid approach to that state. The graph can be found on the appendix section.

As aforementioned, findShortestPath function was obtained from online. There are various of path can be made, and they all have 11 moves (12 states), and they always meet at 3rd and 8th move. This function input the graph, start node, and end node, returns a path in list for output.

The genTrip function input the path generated and return the strings that can be easily read. I have first made a list putting my character definition in a correct order. Then set the direction into two variables either “west” or “east”. Lastly, from the second item in the path, compare with the previous one to locate the difference had been made, and print in a well-organized statement.

[Appendix: The Graph of the Problem]

```
The letters in each node represents the following:
[boat's position, green wife, green husband, blue wife, blue husband, red wife, red husband]

eEEEEEE: {wEEEEWE, wEEWEEE, wWEEEEE, wWEEEWE, wWWEEEE, wWEWEEE, wEEWEWE, wEEEEWW, wEEWWEE}
eEEEEWE: {wEEWEWE, wEEEEWW, wWEEEWE, wWEWEWE}
eEEEEWW: {wWWEEWW, wEEWWWW, wEWEWWW}
eEEWEEE: {wWEWEEE, wEEWWEE, wEEWEWE, wWEWEWE}
eEEWEWE: {wEEWWWW, wWEWEWE}
eEEWWEE: {wEWWWEW, wWWWWEE, wEEWWWW}
eEEWWWW: {wWWWWWW, wEWWWWW}
eEWEWEW: {wWWWWEW, wWWEWEW, wEWWWWW, wEWEWWW, wWWEWWW, wEWWWEW}
eEWEWWW: {wWWWWWW, wWWEWWW, wEWWWWW}
eEWWWEW: {wWWWWWW, wEWWWWW, wWWWWEW}
eEWWWWW: {wWWWWWW}
eWEEEEE: {wWWEEEE, wWEWEEE, wWEEEWE, wWEWEWE}
eWEEEWE: {wWWEEWW, wWEWEWE}
eWEWEEE: {wWWWWEE, wWEWEWE}
eWEWEWE: None
eWWEEEE: {wWWEEWW, wWWEWEW, wWWWWEE}
eWWEEWW: {wWWWWWW, wWWEWWW}
eWWEWEW: {wWWWWWW, wWWEWWW, wWWWWEW}
eWWEWWW: {wWWWWWW}
eWWWWEE: {wWWWWWW, wWWWWEW}
eWWWWEW: {wWWWWWW}
wEEEEWE: {eEEEEEE}
wEEEEWW: {eEEEEEE, eEEEEWE}
wEEWEEE: {eEEEEEE}
wEEWEWE: {eEEEEEE, eEEWEEE, eEEEEWE}
wEEWWEE: {eEEEEEE, eEEWEEE}
wEEWWWW: {eEEWWEE, eEEEEWW, eEEWEWE}
wEWEWEW: None
wEWEWWW: {eEWEWEW, eEEEEWW}
wEWWWEW: {eEEWWEE, eEWEWEW}
wEWWWWW: {eEWWWEW, eEWEWEW, eEWEWWW, eEEWWWW}
wWEEEEE: {eEEEEEE}
wWEEEWE: {eEEEEEE, eWEEEEE, eEEEEWE}
wWEWEEE: {eEEEEEE, eEEWEEE, eWEEEEE}
wWEWEWE: {eWEEEEE, eWEWEEE, eWEEEWE, eEEEEWE, eEEWEWE, eEEWEEE}
wWWEEEE: {eEEEEEE, eWEEEEE}
wWWEEWW: {eEEEEWW, eWWEEEE, eWEEEWE}
wWWEWEW: {eEWEWEW, eWWEEEE}
wWWEWWW: {eWWEWEW, eWWEEWW, eEWEWEW, eEWEWWW}
wWWWWEE: {eEEWWEE, eWEWEEE, eWWEEEE}
wWWWWEW: {eEWWWEW, eWWWWEE, eEWEWEW, eWWEWEW}
wWWWWWW: {eWWWWEW, eEWEWWW, eEWWWEW, eWWEEWW, eWWEWWW, eWWWWEE, eEEWWWW, eWWEWEW, eEWWWWW}
```