My plan for short and long program assignment.

Perform effective actions first, then effective actions, and then continue until there are no remaining actions. If there is one nail left, you win. If there are no remaining actions but still have not won the victory, then undo the action just performed and then try other actions. If you have tried all the steps and no solution has been found, the solution must not exist.

Each solution happens to be 13 moves, because each time you delete 1 nail, and 14 nails start. Therefore, the first solution found is as good as other solutions in terms of length. To identify each hole, I chose to start at 0, then increase from left to right, and from top to bottom to 14. To implement this solution, we need to list all possible transitions on the board. So each jump will be represented by a 3-tuple. The first number is the source hole, the second number is the hole, and the last number is the target hole. The game state can be represented by a set of occupied nails. The list is used to track the actions taken to reach the state of the circuit board.
The constructor takes a parameter called miss to indicate which peg is selected as blank. At the beginning of the game, the peg will be removed from the peg group.

First, I will list the effective actions. If the source hole and the hole to be skipped are occupied, the movement is effective, but the destination is not. To generate a list, all jumped lists will be filtered according to this condition.
Next, I will remove the first two nails and insert a nail at the target location. The movements made will also be added to the movement list. The undo operation is the opposite of the undo operation, and the undo operation is undoes from the list of executed undo operations.
Finally, write the actual solution. It implements the above recursive algorithm. When a win is found, the order of winning moves will be printed, and the search will stop. After creating the class, I can use the following code to solve each starting position. I only need to consider 0, 1, 2, 3, and 4 because other starting positions can be rotated to one of them.

Cb_utils.py
(Legal Move)
def legal_move(board,mov):
“It returns True if mov is a legal move in the board position board, False if it is not”

def legal_move_interface(board_str,mov):
“This function thus serves as the external interface to your legal_move() function.”

(All Legal Moves)
def all_legal_moves(size,board):
“It returns the set of moves that are legal in the board position board. The value of size can be used to obtain the set of all possible moves for the board,”

def all_legal_moves_interface(size, board_str):
“This function converts board_str to your internal representation of a board position, calls

your function all_legal_moves() described above, and returns the value returned by all_legal_moves(). This function thus serves as the external interface to all_legal_moves() function."
def all_legal_moves()
"This function thus serves as the external interface to your all_legal_moves() function."

(Update Board)
def update_board(board, mov):
"It returns the internal representation of the board resulting from making the move mov in board board."

def update_board_interface(board_str, mov):
"This function converts board_str to your internal representation of a board position, calls your function"

def update_board():
"This function thus serves as the external interface to your update_board() function."

In long Program assignment:
cb_solver.py
def cb_one(size, config):
"It returns a string representing a solution to the puzzle represented by the board configuration config (any solution will do); and None if there is no solution."

def cb_all(size, config):
"size is an integer (size ⩾ 4) and config is a string of 0s and 1s, that behaves as follows. It returns a set of strings where each string represents a solution to the puzzle represented by the board configuration config; and set() (the empty set) if there is no solution."