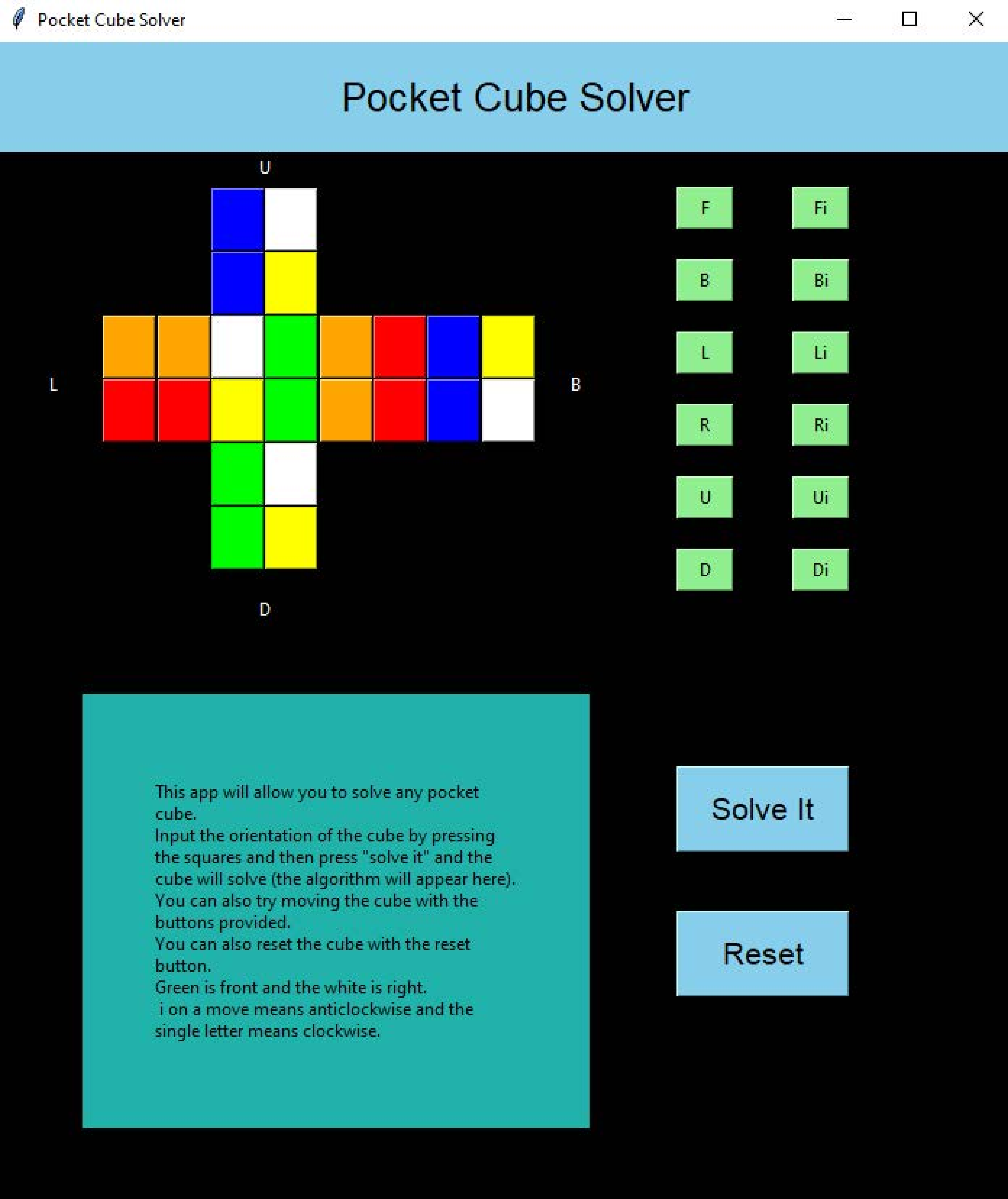
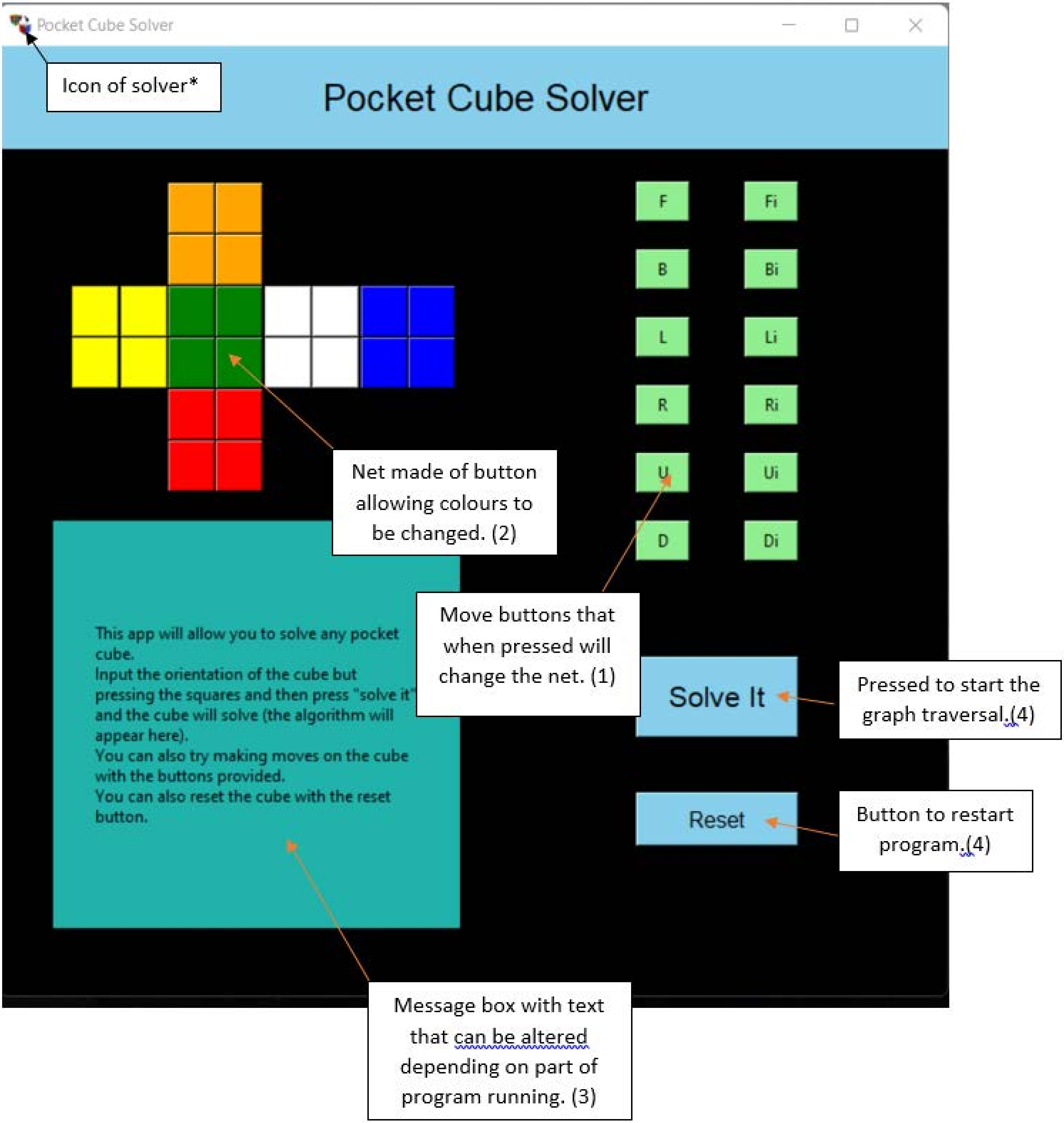


This app will allow you to solve any pocket cube.
Input the orientation of the cube by pressing the squares and then press "solve it" and the cube will solve (the algorithm will appear here).
You can also try moving the cube with the buttons provided.
You can also reset the cube with the reset button.
Green is front and the white is right.
 i on a move means anticlockwise and the single letter means clockwise.

Layout of Interface



This app will allow you to solve any pocket cube.
Input the orientation of the cube but pressing the squares and then press "solve it" and the cube will solve (the algorithm will appear here).
You can also try making moves on the cube with the buttons provided.
You can also reset the cube with the reset button.

Icon of solver*

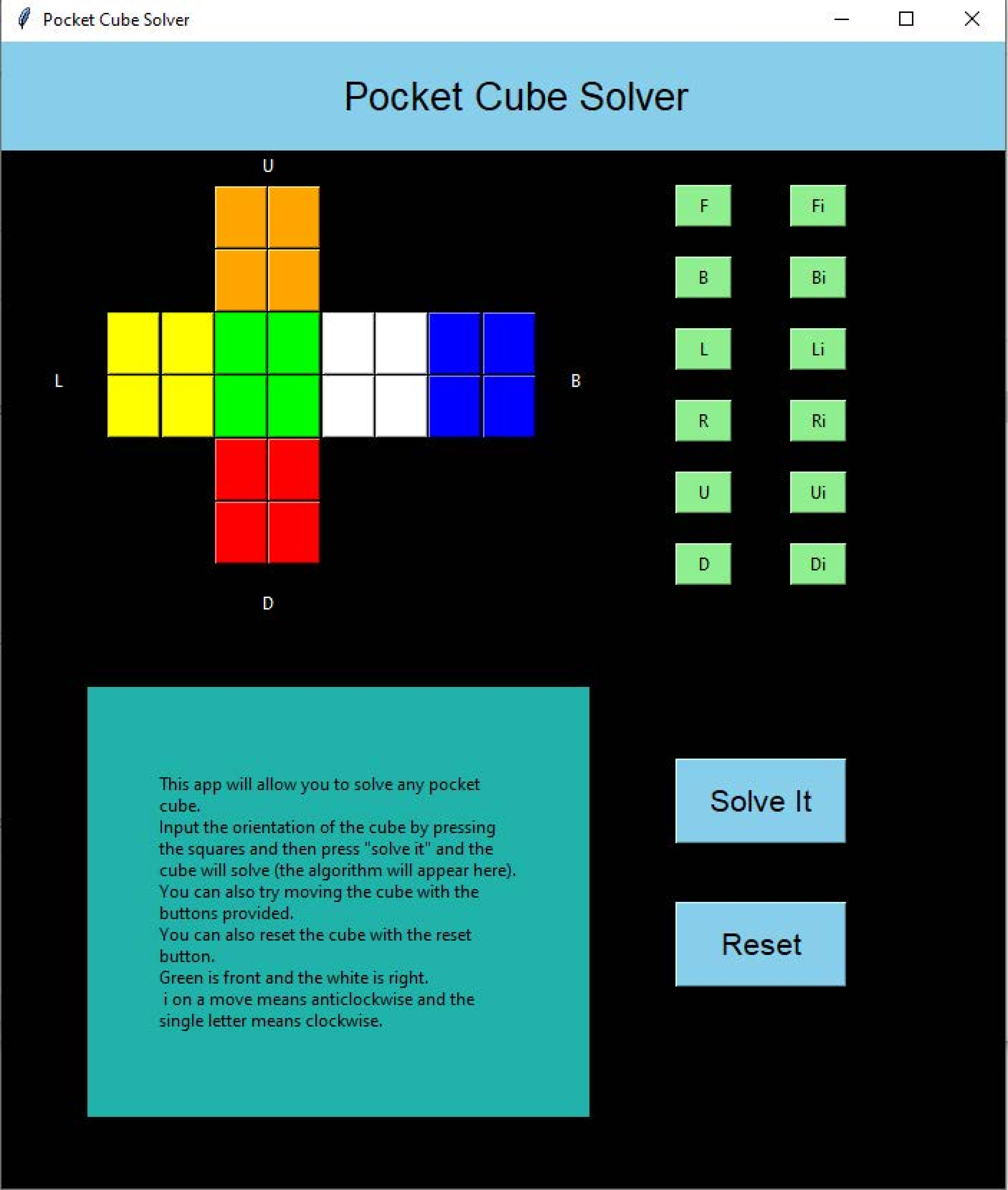
Net made of button allowing colours to be changed. (2)

Move buttons that when pressed will change the net. (1)

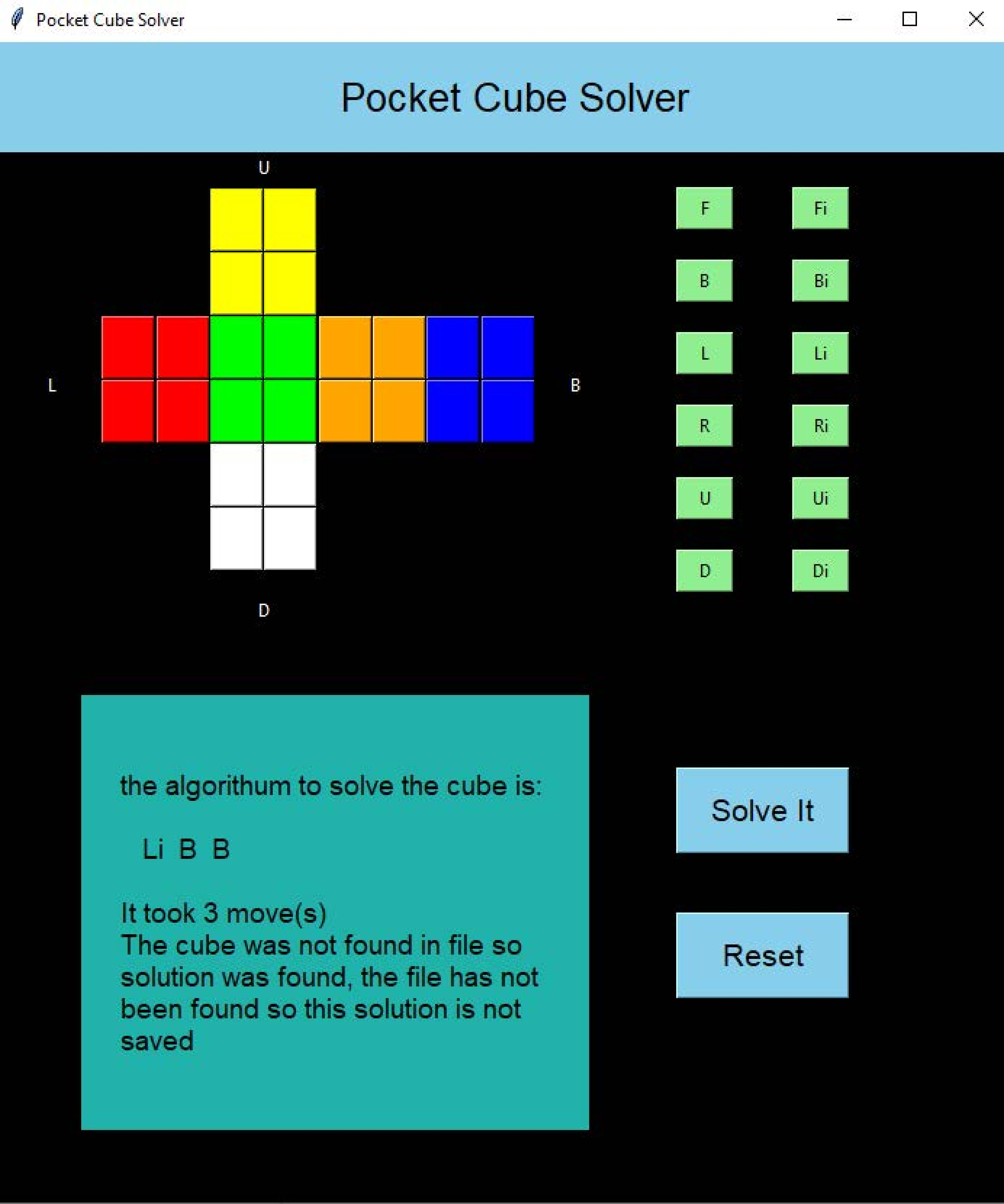
Pressed to start the graph traversal.(4)

Button to restart program.(4)

Message box with text that can be altered depending on part of program running. (3)



This app will allow you to solve any pocket cube.
Input the orientation of the cube by pressing the squares and then press "solve it" and the cube will solve (the algorithm will appear here).
You can also try moving the cube with the buttons provided.
You can also reset the cube with the reset button.
Green is front and the white is right.
i on a move means anticlockwise and the single letter means clockwise.



the algorithum to solve the cube is:

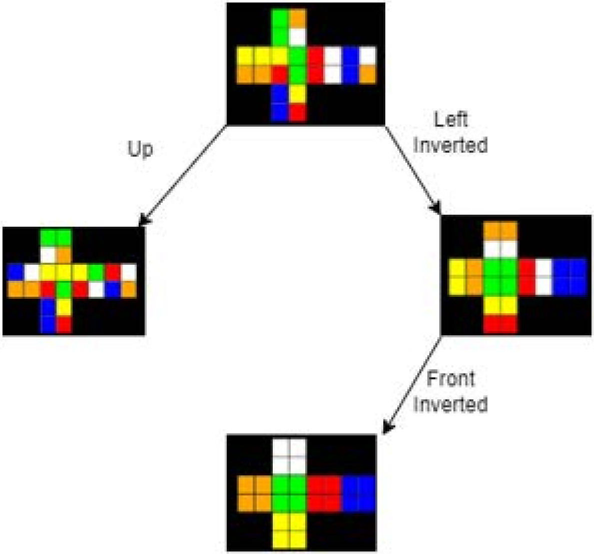
Li B B

It took 3 move(s)
The cube was not found in file so solution was found, the file has not been found so this solution is not saved

Creating the Tree

A tree is made up of nodes and edges. The nodes are the states at that time and the edges are how you get to those states. In the case of solving the pocket cube, the nodes are the orientation of the cube at that time and the edges are the moves performed on the cube.

A very small part of the tree might graphically look something like this:



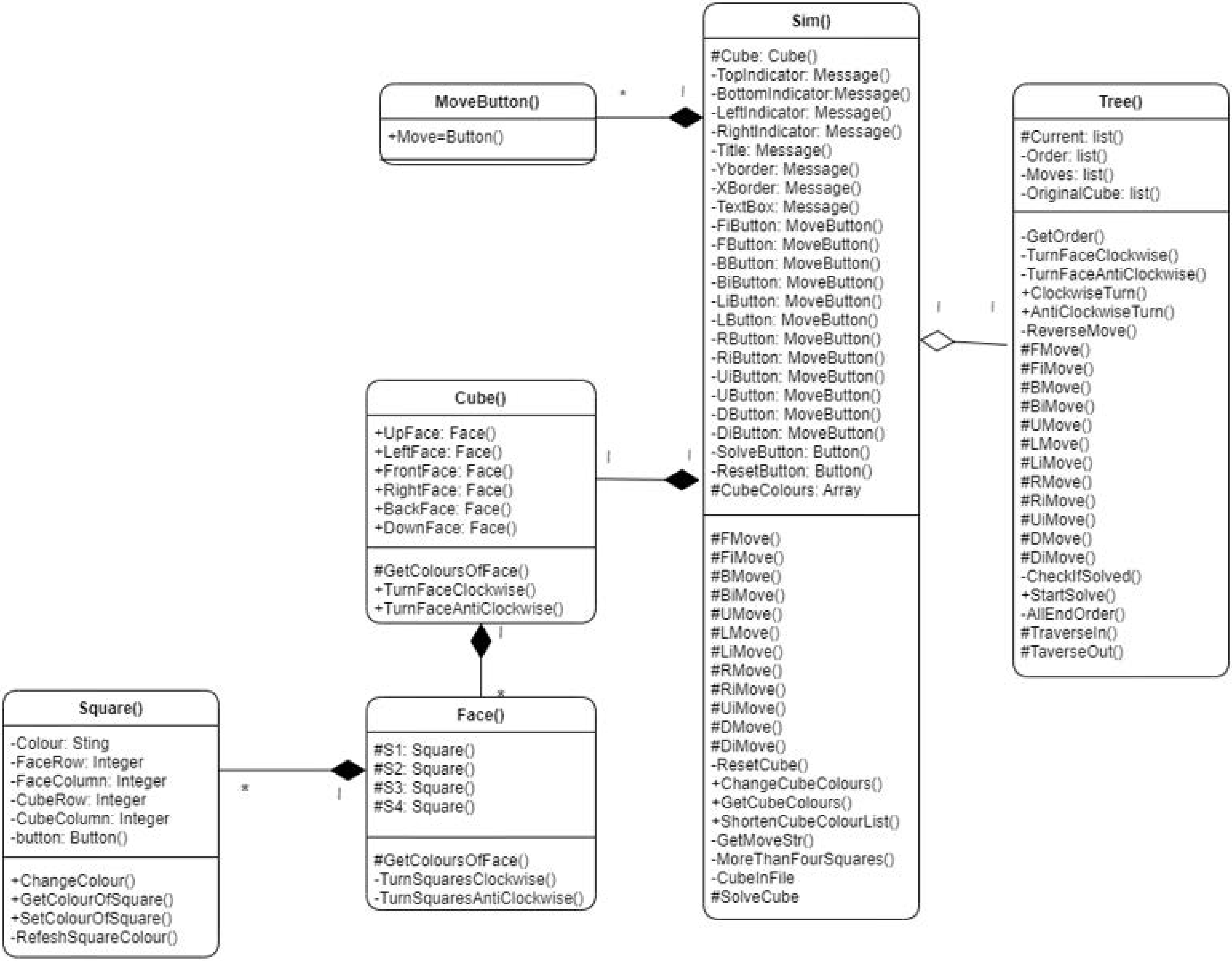
Therefore, the list to solve this cube would be: [Left Inverted, Front Inverted] This would be how the tree is stored,

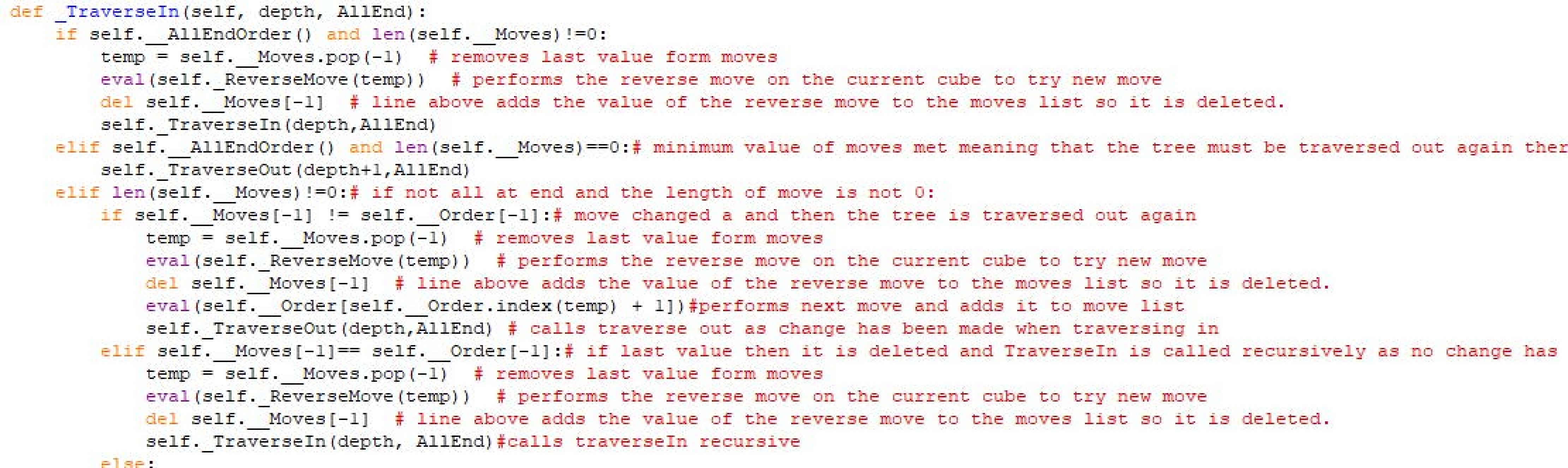
When up was the only move that had be performed, the tree's list would look like this: [Up]

To create the tree, all that is needed is a list of all the possible moves and then these are performed in the traversal in the same order that they are saved in the tree. This will then create a position of a cube that can be saved as that node.

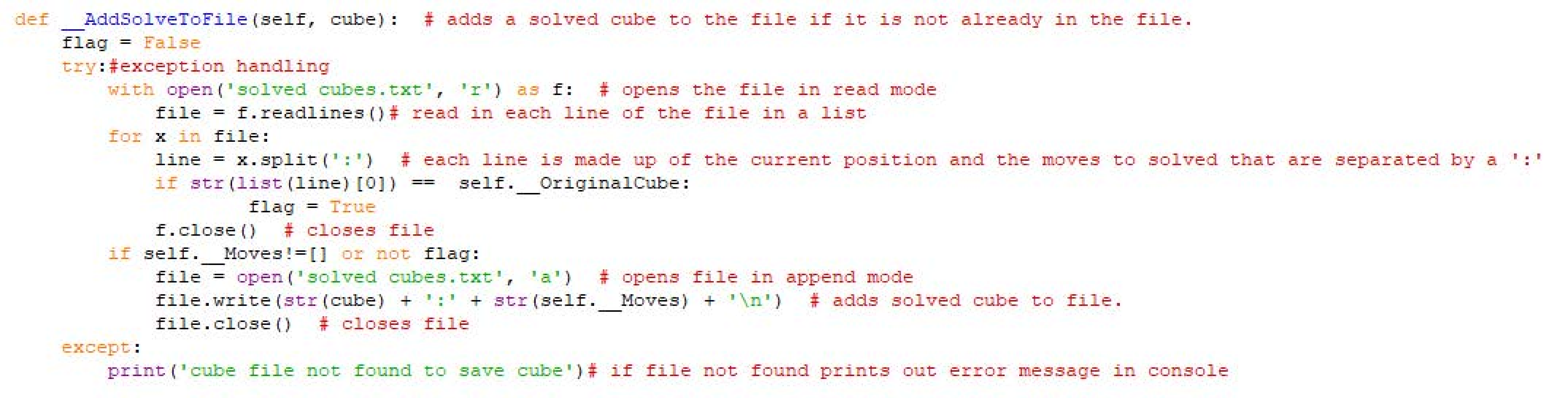
Pseudocode of traversal

```
Private procedure Traverse (depth)
    IF cube solved THEN
        exit traversal
    ELSE IF end of moves does not = end of order THEN
        Remove last value of move and try next in order list
        Traverse(depth)
    ELSE
        IF all values in moves = value at end of order THEN
            Delete  all values in moves
            Add first value of order to moves depth +1 number of  times
            Traverse(depth +1)
        ELSE IF not all values in moves = value at end of order THEN
            Delete values of moves until one found that does not = end of order
            Add first value of order to moves depth number of times
            Traverse(depth)
        ENDIF
    ENDIF
ENDPROCEDURE
```


MoveButton()
+Move=Button()
*
1
Sim()
#Cube: Cube()
-TopIndicator: Message()
-BottomIndicator:Message()
-LeftIndicator: Message()
-RightIndicator: Message()
-Title: Message()
-Yborder: Message()
-XBorder: Message()
-TextBox: Message()
-FiButton: MoveButton()
-FButton: MoveButton()
-BButton: MoveButton()
-BiButton: MoveButton()
-LiButton: MoveButton()
-LButton: MoveButton()
-RButton: MoveButton()
-RiButton: MoveButton()
-UiButton: MoveButton()
-UButton: MoveButton()
-DButton: MoveButton()
-DiButton: MoveButton()
-SolveButton: Button()
-ResetButton: Button()
#CubeColours: Array
#FMove()
#FiMove()
#BMove()
#BiMove()
#UMove()
#LMove()
#LiMove()
#RMove()
#RiMove()
#UiMove()
#DMove()
#DiMove()
-ResetCube()
+ChangeCubeColours()
+GetCubeColours()
+ShortenCubeColourList()
-GetMoveStr()
-MoreThanFourSquares()
-CubeInFile
#SolveCube
1
1
Tree()
#Current: list()
-Order: list()
-Moves: list()
-OriginalCube: list()
-GetOrder()
-TurnFaceClockwise()
-TurnFaceAntiClockwise()
+ClockwiseTurn()
+AntiClockwiseTurn()
-ReverseMove()
#FMove()
#FiMove()
#BMove()
#BiMove()
#UMove()
#LMove()
#LiMove()
#RMove()
#RiMove()
#UiMove()
#DMove()
#DiMove()
-CheckIfSolved()
+StartSolve()
-AllEndOrder()
#TraverseIn()
#TaverseOut()
1
1
Cube()
+UpFace: Face()
+LeftFace: Face()
+FrontFace: Face()
+RightFace: Face()
+BackFace: Face()
+DownFace: Face()
#GetColoursOfFace()
+TurnFaceClockwise()
+TurnFaceAntiClockwise()
1
*
Face()
#S1: Square()
#S2: Square()
#S3: Square()
#S4: Square()
#GetColoursOfFace()
-TurnSquaresClockwise()
-TurnSquaresAntiClockwise()
*
1
Square()
-Colour: Sting
-FaceRow: Integer
-FaceColumn: Integer
-CubeRow: Integer
-CubeColumn: Integer
-button: Button()
+ChangeColour()
+GetColourOfSquare()
+SetColourOfSquare()
-RefeshSquareColour()

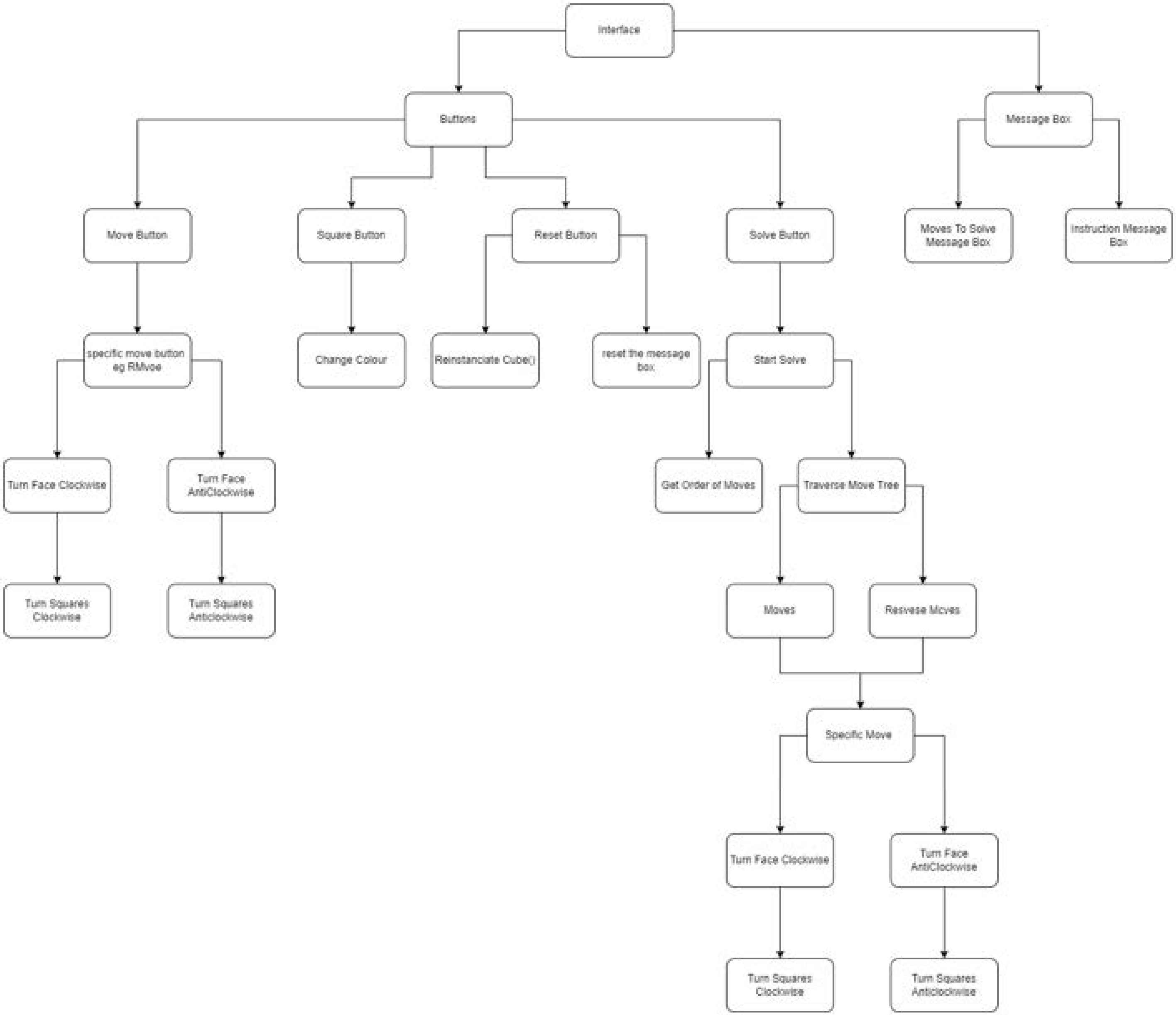


```
def _TraverseIn(self, depth, AllEnd):
    if self.__AllEndOrder() and len(self.__Moves)!=0:
        temp = self.__Moves.pop(-1)  # removes last value form moves
        eval(self._ReverseMove(temp))  # performs the reverse move on the current cube to try new move
        del self.__Moves[-1]  # line above adds the value of the reverse move to the moves list so it is deleted.
        self._TraverseIn(depth,AllEnd)
    elif self.__AllEndOrder() and len(self.__Moves)==0:# minimum value of moves met meaning that the tree must be traversed out again ther
        self._TraverseOut(depth+1,AllEnd)
    elif len(self.__Moves)!=0:# if not all at end and the length of move is not 0:
        if self.__Moves[-1] != self.__Order[-1]:# move changed a and then the tree is traversed out again
            temp = self.__Moves.pop(-1)  # removes last value form moves
            eval(self._ReverseMove(temp))  # performs the reverse move on the current cube to try new move
            del self.__Moves[-1]  # line above adds the value of the reverse move to the moves list so it is deleted.
            eval(self.__Order[self.__Order.index(temp) + 1])#performs next move and adds it to move list
            self._TraverseOut(depth,AllEnd) # calls traverse out as change has been made when traversing in
        elif self.__Moves[-1]== self.__Order[-1]:# if last value then it is deleted and TraverseIn is called recursively as no change has
            temp = self.__Moves.pop(-1)  # removes last value form moves
            eval(self._ReverseMove(temp))  # performs the reverse move on the current cube to try new move
            del self.__Moves[-1]  # line above adds the value of the reverse move to the moves list so it is deleted.
            self._TraverseIn(depth, AllEnd)#calls traverseIn recursive
        else:
```



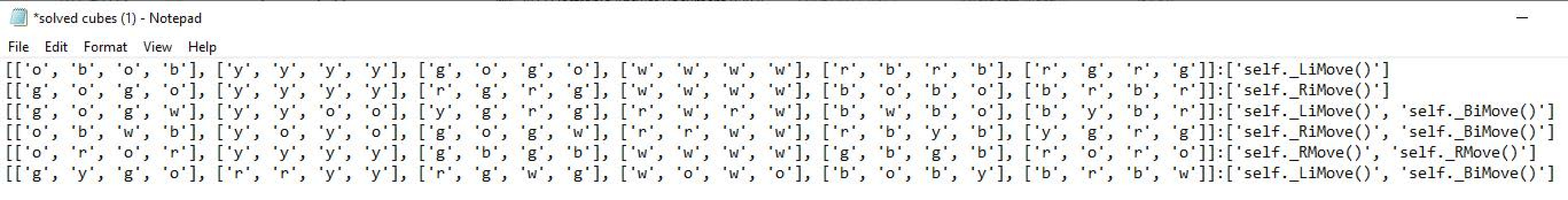
```
def __AddSolveToFile(self, cube):  # adds a solved cube to the file if it is not already in the file.
    flag = False
    try:#exception handling
        with open('solved cubes.txt', 'r') as f:  # opens the file in read mode
            file = f.readlines()# read in each line of the file in a list
        for x in file:
            line = x.split(':')  # each line is made up of the current position and the moves to solved that are separated by a ':'
            if str(list(line)[0]) ==  self.__OriginalCube:
                    flag = True
            f.close()  # closes file
        if self.__Moves!=[] or not flag:
            file = open('solved cubes.txt', 'a')  # opens file in append mode
            file.write(str(cube) + ':' + str(self.__Moves) + '\n')  # adds solved cube to file.
            file.close()  # closes file
    except:
        print('cube file not found to save cube')# if file not found prints out error message in console
```

Hierarchy chart of system

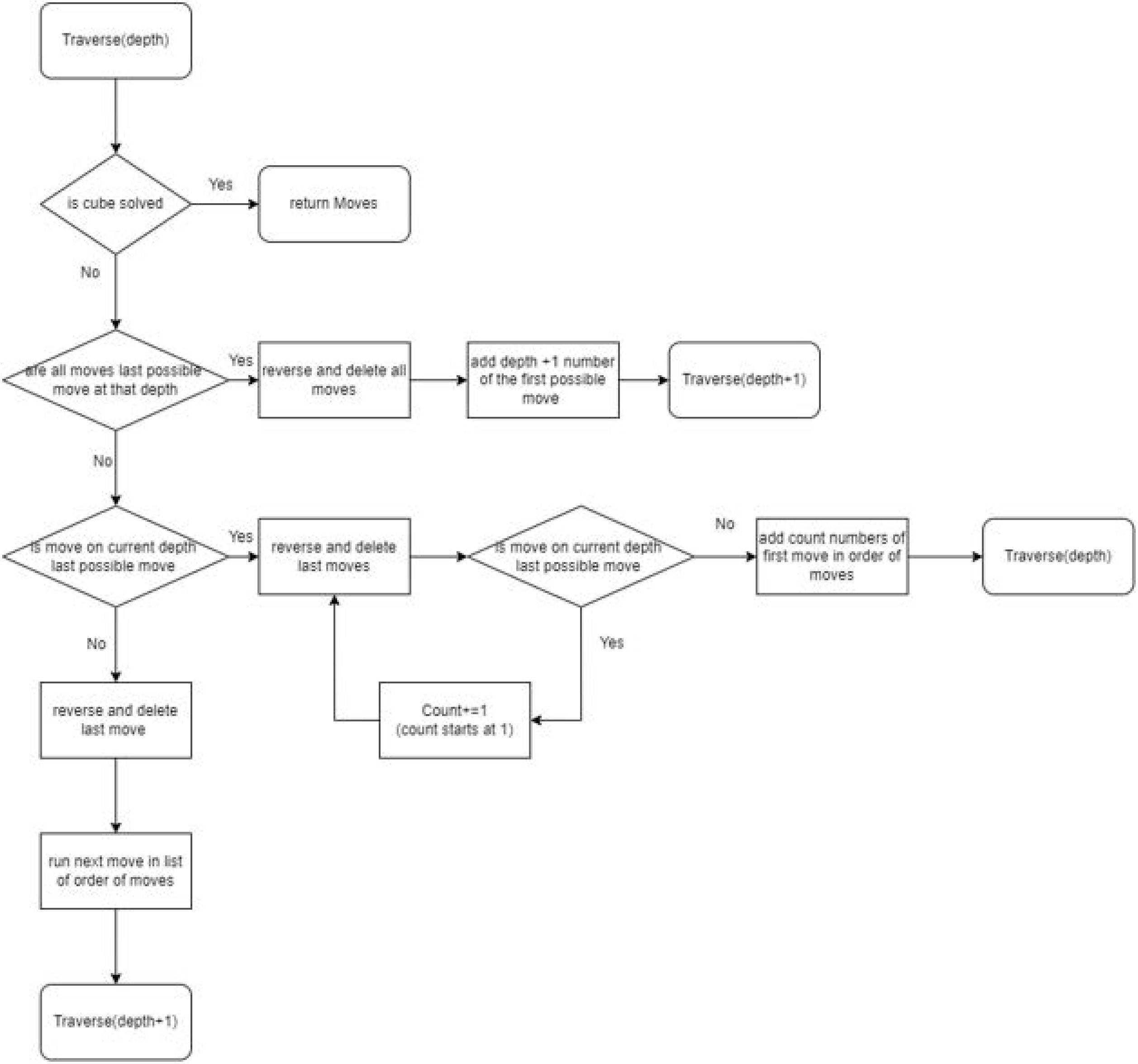


Tests to produce

Test no	Objective no	Purpose of test	Test type	input	Expected result
1	1	To ensure all buttons on the net work	Typical	Press each button on net 6 times noting the colours that the squares become each time	For each square to change colour to the next colour
2	2	To find out how the program stores the cube's position	Typical	none (Added print line to code to output data type)	A 2D list of the colours of each square on each face
3	3	To ensure that the program rejects an impossible cube pattern	Erroneous	Press top left button on white side of the net to make an impossible cube pattern	An error message saying it cannot be solved
4	3	To ensure that the program rejects an impossible cube pattern	Erroneous	Press bottom right button on red side of the net to make an impossible cube pattern	An error message saying it cannot be solved
5	3	To ensure that the program accepts a possible pattern	Typical	Create acceptable position by entering the position of a cube that is off by one move. Do this by pressing FI button	For the cube to be accepted and continue to solve the cube



```
[['o', 'b', 'o', 'b'], ['y', 'y', 'y', 'y'], ['g', 'o', 'g', 'o'], ['w', 'w', 'w', 'w'], ['r', 'b', 'r', 'b'], ['r', 'g', 'r', 'g']]:['self._LiMove()']
[['g', 'o', 'g', 'o'], ['y', 'y', 'y', 'y'], ['r', 'g', 'r', 'g'], ['w', 'w', 'w', 'w'], ['b', 'o', 'b', 'o'], ['b', 'r', 'b', 'r']]:['self._RiMove()']
[['g', 'o', 'g', 'w'], ['y', 'y', 'o', 'o'], ['y', 'g', 'r', 'g'], ['r', 'w', 'r', 'w'], ['b', 'w', 'b', 'o'], ['b', 'y', 'b', 'r']]:['self._LiMove()', 'self._BiMove()']
[['o', 'b', 'w', 'b'], ['y', 'o', 'y', 'o'], ['g', 'o', 'g', 'w'], ['r', 'r', 'w', 'w'], ['r', 'b', 'y', 'b'], ['y', 'g', 'r', 'g']]:['self._RiMove()', 'self._BiMove()']
[['o', 'r', 'o', 'r'], ['y', 'y', 'y', 'y'], ['g', 'b', 'g', 'b'], ['w', 'w', 'w', 'w'], ['g', 'b', 'g', 'b'], ['r', 'o', 'r', 'o']]:['self._RMove()', 'self._RMove()']
[['g', 'y', 'g', 'o'], ['r', 'r', 'y', 'y'], ['r', 'g', 'w', 'g'], ['w', 'o', 'w', 'o'], ['b', 'o', 'b', 'y'], ['b', 'r', 'b', 'w']]:['self._LiMove()', 'self._BiMove()']
```


Traverse(depth)
is cube solved
Yes
return Moves
No
are all moves last possible move at that depth
Yes
reverse and delete all moves
add depth +1 number of the first possible move
Traverse(depth+1)
No
is move on current depth last possible move
Yes
reverse and delete last moves
is move on current depth last possible move
No
add count numbers of first move in order of moves
Traverse(depth)
Yes
Count+=1 (count starts at 1)
No
reverse and delete last move
run next move in list of order of moves
Traverse(depth+1)