My initial plan was to build an abstracted, and well-organized Program. However, I put too much in over my head, as this project is my first major experience with C#. In the end, I decided to give myself the benefit of the doubt to make a simpler program, essentially in one file- and to later adjust and refine the program to function entirely with an object oriented design later down the road.

There are two essential classes at the moment:

* GameBoardSolution
* MainClass

GameBoardSolution holds three variables: int row, int col, UInt32 previous, and two constructors.

Row and Column each signify the move that was made to create the change between the previous step, and the current position. Previous is the Unsigned Integer (32 bit) value that is equivalent to bit-string representation of the gameboard that created the current GameBoardSolution with the move from (row, col).

MainClass controls the game itself.

This class is where the heart of the program takes place. There are 6 functions separate from the Main() function.

* Convert()
* ConvertBack()
* Flip()
* IterateSolutions()
* DisplayBoard()
* Find()

Convert- Takes parameter: UInt32 board

UInt32 board is a representation of the gameboard as an Unsigned integer- and needs to be converted into a Boolean array for the Flip and DisplayBoard fuctions to properly process the data in *board.* Returns bool[,]

ConvertBack- Takes parameter: bool[,] boardRepresented

Bool[,] boardRepresented is the array that in order for storage purposes, needs to be converted back into an Unsigned Integer (32 bit) for *IterateSolutions* and *Find* need to search/process boards. Returns UInt32.

Flip- Takes parameters: UInt32 *board*, int *row,* int *col*

Flips all the pieces in the *row,* and *col*umn on *board.* Modifies *board*, and doesn’t return any data.

IterateSolutions- Takes parameters: Dictionary *gameSolutions*, Queue *incomplete*

Pulls boards out of incomplete, opens all children, and places previous(parent) boards into the Dictionary. If a child board is a duplicate, or already processed board, then the duplicate is ignored. Does not return.

DisplayBoard- Takes: UInt32 *boardValue*

BoardValue is the unconverted board, which is then converted and printed to the terminal. Does not return.

Find- Takes: Dictionary *gameSolutions*, UInt32 *search*

Checks the dictionary for any GameSolution that holds the key *search*. If there is a solution, then it will find the path for the solution from the key to the goal state.

Currently, the MainClass uses a retrograde analysis of the Game Board, compiling all the possible boards into a dictionary, and taking a few minutes to complete. I found that rather than 225 possible solutions, there exist only 217 solutions. The gameboard itself cannot have moves that exist beyond the 217 positions. I would like to believe that my solution could scale just as well, although likely still exponentially should there be a bigger board. Due to the amount of possible positions simply growing exponentially by adding rows and columns, the time taken will also increase exponentially. I think my algorithm works well for the cases provided, however, with a larger board and more adjustments, the program itself can run more efficiently.