AE552 Midterm - Akshat Dubey (akshatd)

Code organization

The code is split into 3 files

- `chessboard.h`:
  - contains the `ChessBoard` class definition, which holds the state of a chessboard when traversing through the search tree
- `chessboard.cpp`:
  - contains the `ChessBoard` class implementation
  - contains a helper function to print the chessboard to std::cout for a better user experience
- `midterm_main.cpp`:
  - contains the main function handling tests and user input
  - contains all the simple recursive functions needed for the iterative deepening search

Agorithm Description

The most important part of the code lies in the function `ChessBoard::addQueen`, which handles adding a new queen to the board, and checks if the queen can be added or not by checking the row, column and diagonals on the board.

The algorithm follows from the example we went through in class, where we go column by column and see if a queen can be placed anywhere for a given state of the board.

In the main function, the `solveNQueens` function sets up the recursive search by calling `findSolutions` for all rows in the first column.

The recursive calls inside `findSolutions` are relatively simple, the base case for the recursion to stop is if the desired depth has been reached or if we have as many queens as we need. The function then copies the board that has been passed in for each row and then tries to add a queen in the particular row. If adding a queen was successful, it calls itself again to continue the recursion. The copying is necessary to keep track of multiple solutions branching out from the same root recursive call.

Once a solution is found, it is added to the `solutions` vector, which is then returned by `solveNQueens` to the main function, containing all the solutions for the given chess board size.

Results and Discussion

A couple of main points to note:

- The iterative deepening search only works when the depth = size of the board, since we have to search `n` columns to place `n` queens.

- The program has tests that verify that the number of solutions are correct for any given n to make sure there are no regressions when optimizing the logic.
- The algorithm can be optimized further if we keep track of all the spots on the chessboard that have already been visited by parent recursive function calls and might help when the size of the board becomes large.