Salvy Cavicchio
N10995724

Instructions to play game:

OPTION 1:

1) Open the ZIP file.
2) Place the file a.out into your Desktop.
3) Open the terminal or command line of your computer
4) Navigate to your Desktop folder in the terminal ("cd Desktop" on Mac)
5) Run the a.out file. (command: "./a.out" on Mac)

OPTION 2:

1) Open the ZIP file and place the main.cpp file on your Desktop.
2) Use the terminal or command line of the computer to compile the main.cpp file. (on mac this command is usually "make foo.cpp")
3) Run the output file you created while compiling.

OPTION 3: (WINDOWS)

http://smallbusiness.chron.com/execute-cpp-file-windows-28480.html

A CPP file contains C++ programming code you must compile before you can run the file's code on your Windows system. You use Visual Studio to compile the CPP code on Windows. The compiling process creates an EXE file, which is an executable that runs on a Windows computer. CPP files are typically distributed in sample C++ programs, so you can view the code, compile the app and review the results.

1. Click the Windows "Start" button and select "All Programs." Click "Microsoft .NET Express," then select "Visual Studio Express" to open the software.

2. Click the "File" menu item, then select "Open." Double-click the CPP file to load the source code in Visual Studio.

3. Click the "Build" menu item and select "Build Solution." The software compiles the code and creates the EXE file.

4. Select the "Run" button. The compiler runs the executable file, so you can review the results of the C++ code.

OPTION 4: (MAC)

1) Install X-Code and open AI_11.xcodeproj
2) Do "COMMAND-R" to run the program.

Salvy Cavicchio
N10995724

DESCRIPTION OF CODE

The code uses a class gameBoard() to represent the board. The board uses five bitsets to represent the various pieces on the board. The five bitsets represent the location of white pieces, black pieces, the white castles, the black castles, and the legal board spots. The program generates moves by first checking to see if there are any capturing moves that can be performed by the user, and then if there are no capturing moves it will search for “simple” type moves (aka: anything non capturing). The program takes user input to move pieces through the command line, converting an input of row and column (example: C4) to an index in the bitset from 0 to 111. When it is the AI turn, the move generation will remain the same and the AI will call the alpha beta function to create an evaluation for each game state possible with each piece.

Bitsets were chosen to represent the board so that I could take advantage of the processing speed of bitset operators such as AND, NOT, and OR. These operators are used heavily in the functions that check to see if a move is valid (this is done by checking if(legalMoves&whitePieces == legalMoves|whitePieces), and also in the functions that check if a player has won the game (by performing if((blackPieces&whiteCastle).any==1). Bitset math is very efficient and takes 8 times less memory space than a similar boolean array.

There are different levels of difficulty that a player can choose in the start of the game. This difficulty directly corresponds to the maximum depth that the alpha beta function will search. The max depth is set to 1+the difficulty level the player chose. This program was able to completely search depth level 5 of this program in an average time under 3 seconds. This corresponds to up to 1 million nodes in the required 10 second time. The player is allowed to choose a depth level greater than 4 (difficulty greater than 5) but is warned before the program runs of the expected computation time. I believe the speed of this program is very impressive compared to similar programs (an estimate of 4 times faster than the programs I compared it to). I attribute this to the use of bitsets and programming in c++ rather than python.

The evaluation function used takes into account the number of pieces a player has left on the board, the sum of the distances of each piece from the castle, and the distance of the closest piece a player has to a castle. It also looks to see if there is the possibility of a single piece of any color being in the enemy castle. The number of pieces a player has is important because it directly correlates to the number of moves they will be able to perform. The weight of the evaluation function is more placed on the distance of the closest piece a player has to a castle so that the program will attempt to become “safe” by placing a close piece in the castle rather than moving a “second” piece that is farther away. The most influential factor of the evaluation function is having a piece in the enemy castle. This value is made so large that a move that leads towards having this outcome receives a far higher score than most other moves.