1. **Team Information**

**Team name:** Deshawn Dana

|  |  |  |
| --- | --- | --- |
| **Member**  **(alphabetical by last name):** | **Email address:** | **Percentage contribution:** |
| Justin Buerano | justinbuerano@gmail.com | 16.67% |
| Deshawn Dana | Dashwen@gmail.com | 16.67% |
| Diana Galvan | Dianag118@csu.fullerton.edu | 16.67% |
| Alberto Gomez | otrebla117@csu.fullerton.edu | 16.67% |
| Clara Kim | klayrah\_kim@csu.fullerton.edu | 16.67% |
| Eric Roe | ericroe@csu.fullerton.edu | 16.67% |

1. **Description of Program Organization (Modularization)**

**Modules:**

1. **chessmain.py**: The main file which combines the other modules listed below. This module contains the functions as outlined in this project: setupBoard(), play(), heuristicX(), heuristicY(), move(), and showMove(). It also has the minimax algorithm functions, and functions for other tasks we saw as necessary or convenient.
2. **chess from python-chess** (github OSS): The chess engine is imported from python-chess. In chessmain.py, we interact with an instances of the board class throughout our program.
3. **LogInterface class in log.py**:Log functionality is handled through our LogInterface class. In chessmain.py, we interact with an instance of the LogInterface class to read/write as needed.
4. **Strategy class in strategy.py**: This module holds the board layouts which we refer to within our heuristic functions to decide what move to make.
5. **FenParser class in fenparser.py**: This module is from another author, which we have credited. This class is used for parsing the popular FEN format for storing the chess board (a single string), and allowing us to index it by row and column. This is helpful for tracking down the location of individual pieces.
6. **Description of Strategies**

**Strategy:**

We created an 8x8 grid associated with chess piece and assigned values to each square in the grid. These values indicate what a good location is for the piece on the board. For the king, the middle of the board is preferred as it gives the most escape routes. For the Knight (Night), the location is prioritized based on how many moves can be made. This is also middle-centric. The rook seeks control of the 7th rank generally, since the board is in the late-game. Additional strategy methods include evaluating the current board layout to see if we are in checkmate, stalemate, or have a threefold repetition, so they may be avoided. Then we check the pieces on the board to see if the current board configuration has them. Losing a friendly piece is bad, and killing an enemy piece is good. The points are added or subtracted based on these conditions, and the final value is returned.

**Heuristic X Pseudocode:**

''' The heuristic for X. Try to optimize X's move. '''

# Input:

# @param A single board state (based on a potential move)

# Output:

# The value of the potential move.

def heuristicX(state):

# The value is initialized as 0

value = 0

# Put the board into FEN format

board = fen\_format\_of\_(state)

#find King Value

row, col = find\_the\_coords(board, 'K')

value += value\_from\_king\_strategy(row,col)

# Check the condition of the board and reduce score if any bad conditions are met

if state is checkmate:

value -= 10000

elif state is stalemate:

value -= 30

elif state can claim threefold repetition():

value -= 50

# Find rook value

try:

row, col = find\_the\_coords(board, 'R')

value += value\_from\_rook\_strategy(row,col)

except:

# If the piece isn’t found, it’s bad

value -= 20

# Find night value

try:

row, col = find\_the\_coords(board, 'N')

value += value\_from\_night\_strategy(row,col)

except:

# If the piece isn’t found, it’s bad

value -= 5

# Find enemy night value

try:

row, col = find(board, 'n')

except:

# If the piece isn’t found, it’s good

value += 20

# Return the calculated value of the potential move

return value

**Heuristic Y Pseudocode:**

''' The heuristic for Y. Try to optimize Y's move. '''

# Input:

# @param A single board state (based on a potential move)

# Output:

# The value of the potential move.

def heuristicY(state):

# The value is initialized as 0

value = 0

# Split the board into row and column for cross-referencing with our strategy

board = fen\_format\_of\_(state)

# Find king Value

row, col = find(board, 'K')

value += get row and column of King

# Check the condition of the board and reduce score if any bad conditions are met

if state is check():

value -= 10

elif state is checkmate():

value -= 10000

elif state is stalemate():

value -= 30

elif state can claim threefold repetition():

value -= 50

# Find night value

try:

row, col = find(board, 'n')

value += value\_from\_night\_strategy(row,col)

except:

value -= 20

# Find enemy rook

try:

row, col = find(board, 'R')

except:

# If the piece isn’t found, it’s good

value += 50

# Find enemy night

try:

row, col = find(board, 'N')

except:

# If the piece isn’t found, it’s good

value += 15

# Return the calculated value of the potential move

return value

**Data Structures:**

**Search Data Structure:**

The data structure for holding the moveset which should be searched through is a list. Through the python-chess module, we generate a list of the valid moves and store them in a list. Because of the recursive nature of the Minimax Algorithm and the way Alpha-Beta pruning works, a list is all that was needed to retain a proper “tree” structure of board states. The list of moves are searched through in order, the states are evaluated, and the pruning optimizes the strategy.

**State Data Structure:**

The data structure we use to represent a state is the python-chess module’s built-in board class. This was the optimal choice for us because it allows us to interact with it as we need, including changing it into a string (the common FEN format in chess) and breaking that further into a list of lists. A single state is therefore a single board as described by the board class.

1. **Summary of Game Results**

When playing against ourselves, we always won (and we always lost). Playing chess against yourself is sad, so we played against the other teams in our group. Here are our game results:

**Opponent:** Reboot

**Result:** Win

In game 1, we were X. We won because their program crashed due to a runtime error after 23 moves (it tried to put a piece on top of their own piece). In game 2, we were Y and we stalemated after 32 moves. Overall, this was a win because they automatically lost game 1, and we tied game 2.

**Opponent:** Scorpions

**Result:** Win

In game 1, we were X and we stalemated in 20 moves. In game 2, we were Y and we stalemated after 11 moves. Both teams agreed that stalemating in fewer moves while on “defense” as Y was a win, so we were deemed the winner.

We won against both teams in our group. Now to sweep the finals.

1. **References**

* 1. <https://github.com/niklasf/python-chess>, **The Python Chess Engine we used,** coded by a community project headed by niklasf (more information can be found on the link).
  2. <https://github.com/tlehman/fenparser>, **A FEN Parser,** coded bytlehman.

1. **Comments or Lessons Learned From This Assignment**

**Justin:**

One major lesson I learned from this assignment was how difficult it is to implement a strong heuristic and especially if given a time limit for each move. Another lesson I learned was the importance of well documented code. Once we, as a team, decided on an open source chess engine, it took a good amount of time in order to figure out what the various functions did.

**Deshawn:**

A lesson that I’ve learned from this assignment was how to modularize each of the pieces of our requirements together. I felt that was one of the reasons why our group had a slow start, more specifically breaking down what each of the functions should do, should not do, etc. Having played chess before, I found it interesting how FEN was used as a standard to represent different board states. It made integration of the python chess library manageable and allowed for us to progress towards a working implementation of the assignment. Finally, I have learned the difficulties of maintaining communication across multiple members. Given that our schedules did not match at times, keeping everyone on the same page was crucial to our success towards the project.

**Diana:**

One of the lessons that I learned from this assignment was how to play chess, or at least the basic concepts of how to play chess. I have never played it before but my teammates were patient enough to give me a simple breakdown of how the king, rook, and knight should move. Another thing that I learned was how to work with my teammates even though we all have different schedules, and the six of us couldn’t always meet up at the same time.

**Alberto:**

I learned how difficult it can be to design a working heuristic function, as well as implementing a minimax algorithm. It was a challenge to discover a way to represent each state in a tree while still being able to interact with the python-chess API that our group used in the assignment. I learned that finding a useful API can benefit a project immeasurably. Not having to code the rules and mechanics of chess by hand was a godsend considering the limited amount of time in which our group was able to meet. Lastly, I learned how challenging it can be to find a time when 6 people, with different jobs and varying schedules, might all meet at the same time.

**Clara:**

One of the lessons that I learned in this assignment was learning how to work in a team. With five other members in our team, there were many times where we had to come together and figure out what we wanted and didn't want. At times it did get confusing, but in the end we were able to delegate the jobs accordingly and bring all of our ideas into one working code. Another lesson that I learned while working on this project was being able to actually see and develop code to a topic that we have learned in class. I was always used to seeing such algorithms written out either in diagrams or in pseudo code so being able to see the program actually generating moves by using the heuristic that we developed was very interesting to see.

**Eric:**

I have learned more about chess than I would like to know. I learned about the different board formats in chess, the different move storage formats in chess, and the threefold rule for calling a stalemate. I had no idea that rule existed before this class. But more importantly, I learned the intricacies of how the Minimax Algorithm with Alpha-Beta pruning can parse through potential game states and prune the board to greatly reduce the number of states evaluated, thus reducing the search time. Seeing it in class is one thing, but implementing it was challenging and rewarding. I learned how to use github more effectively to share code updates with my peers, and I learned more about pair (and sometimes triplet) programming. When multiple people are troubleshooting a bug, the solution can be found a lot faster. I wish I always had this option.