(a) Team Name: Victorious Secret

Team Member Names:
Adam Beck adamjbeck@csu.fullerton.edu 50%
Art Grichine artgrichine@csu.fullerton.edu 50%

(b) How to run the program:

Computer information:
Operating system tested on: Mac OSX Yosemite
Python version: Python 3.4

Navigate to the 'Code' directory if you are running it from your terminal
run program with following command:

python34 Play.py

You'll be asked if this is a test. Entering 'y' will continue and ask you how many moves to simulate (each move counts for one move for player X and one for player Y i.e. 2 turns). Entering the amount of moves to be simulated will trigger the program to open the file called 'testCase.txt' and run the board configuration. Board configuration in test case must follow the sample provided in the assignment as follows:

x.K(5,6), x.R(8,5), y.K(6,8)

When the program is finished, a file called 'gameResult.txt' will be created and the final board position and result will be printed inside the file as well as the command line.

Entering 'n' to the question 'Is this a test?' will result in player vs. player mode which is created for the class competition.

Note: The file used to import the board configuration will be re-written to accommodate playing against another program. This is done so that a program can read from a file, play a move and then re-write the file thereby allowing another program to do the same until a conclusion is reached. The problem with this is that if the program crashes, the file that was used to import is erased. After this, re-running the program will do no good due to no data on the input file. If program crashes, make sure 'testCase.txt' contains board configuration. This program did not crash during the test cases however we felt it was necessary to mention.

A brief 1 move run:

(c) Brief description on the algorithm and heuristic function(s) implemented:

The algorithm in Minimax.py takes the parameters board and turn and returns a chosen move. Within the module Minimax.py are a controller function called minimax() and per the project requirements two functions (MIN and Max) called heuristicY() and heuristicX(). These functions take the parameters board, depth, turn, alpha, beta. The functions were built originally as pure minimax functions and then converted to implement alpha-beta pruning. The heuristicY() and heuristicX() functions each return an evaluated value back to the minimax controller function. Based on the heuristic value received, the maximized or minimized value is chosen by the minimax controller based on the turn of the player (playerX maximizes while playerY minimizes their turn). The minimax controller function then choses the corresponding move to the chosen 'best' score and returns that move to the Play.py module.

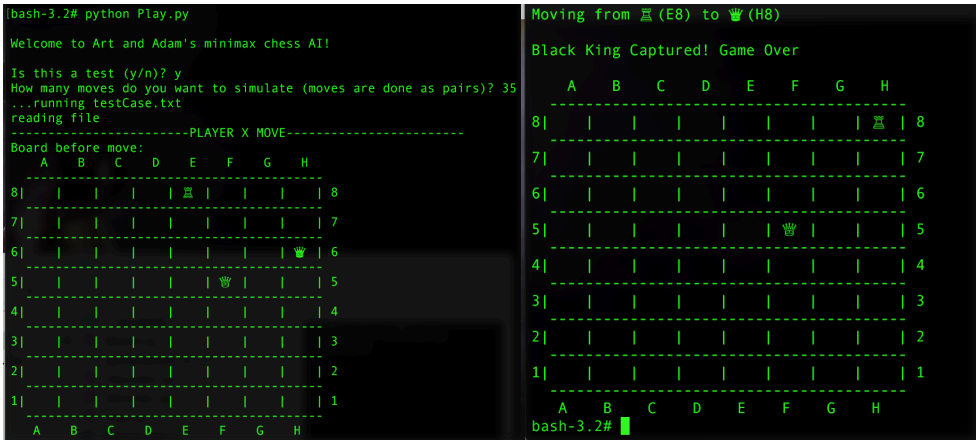
The heuristic or 'evaluate move' functions in this program are comprised of two parts. The evaluate_move() function checks for possible piece capture, check, and checkmates. If found, a large bonus/cost is assigned to the move based on the given piece. These rewards are indifferent to the strategy/heuristic used therefore they are calculated outside of the Heuristic module. Also, a file called Heuristic.py contains a two dimensional array which resembles a board to give

bonus points to where a piece would gravitate to. Values are set higher for areas where it would benefit a rook to be such as one square removed from the end/beginning of a row or column (e.g. A2, A7, G8, H7, etc.). Also, cost is added for a rook to be in a poor offensive position such as the very corner (e.g. A1, A8, H8, H1). Likewise, the king has a similar array which adds bonus to gravitate to the middle of the board to avoid getting trapped. Lastly, the module also checks for possible check scenarios and assigns cost/reward depending on the move and player.

Game result for each test case:

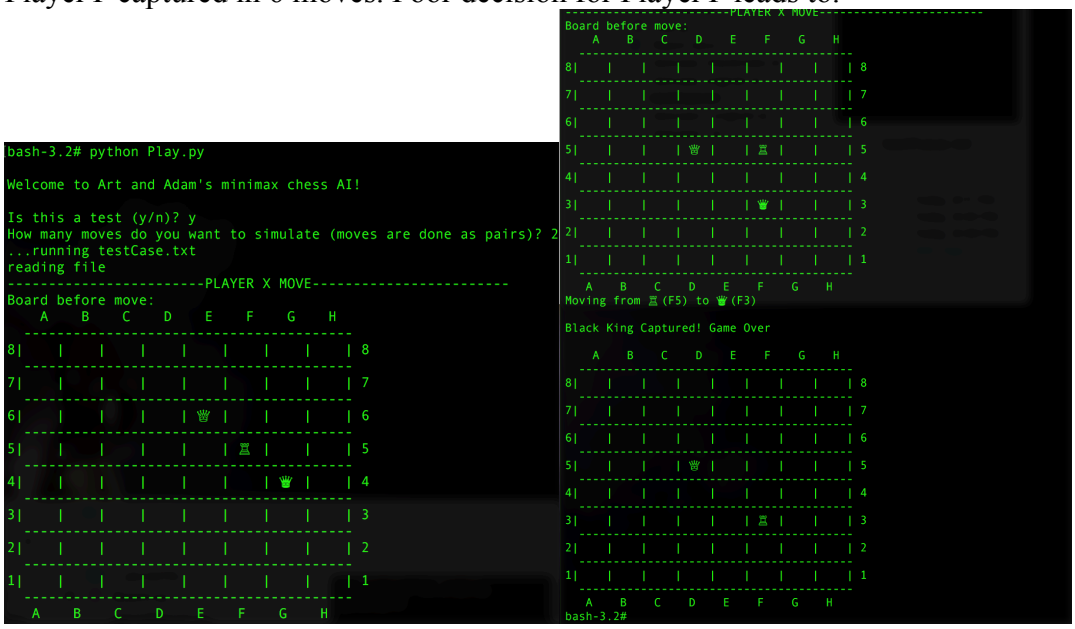
Test case1: x.K(5,6), x.R(8,5), y.K(6,8)

PlayerX (White) captures PlayerY (Black) king in 8 moves.



Test case2: x.K(6,5), x.R(5,6), y.K(4,7)

PlayerY captured in 6 moves. Poor decision for PlayerY leads to.



Note: If player makes poor decision their piece will be captured and victory declared.

Note: Checkmate case is possible however did not happen from given test cases:

(d)
References:
https://chessprogramming.wikispaces.com/minimax
https://chessprogramming.wikispaces.com/Alpha-Beta
http://www.naftaliharris.com/blog/chess/

(e)
Optional comments ~
Creating new instances of our Board class during the mini-max algorithm proved challenging in Python. Rather than making a new object, python would overwrite our current board during new instance creation. In Python, the deepcopy() function must be used from the 'copy' library. This also requires that instances within our board be deep copied as well. This proved to be challenging and time consuming however we eventually solved this problem by applying deepcopy() to all parts of our object Board. In retrospect, aspects of this project such as minimax and object management could have been easier in a language such as C++ or C#.

It was difficult to come up with an effective heuristic function. The minimax algorithm itself if fairly simple however as we discussed in class, the heuristic function is what makes the program effective. With a game like chess, a human chosen heuristic seems ineffective and inefficient when considering how to award a player outside of just determining attack moves. We have determined that choosing an effective heuristic for a full chess game would be very difficult.