Breakthrough

Programming Assignment 1

By Christopher Lebovitz and Dylan E. Wheeler

Date:	2019 10 18
Course:	CSC 412 - Introduction to Artificial Intelligence
Instructor:	Dr. Bo Li

Abstract

This program (`Breakthrough.py`) uses an AI to play the game Breakthrough. It uses a number of strategies and decision making processes to select moves, and alternates between two automated players until one of them reaches a win condition. On each player's turn, a move is selected from a search tree built of all possible states from the current state of the board.

The program uses four strategies and two search algorithms. Each player may use one of the four strategies and one search algorithm. Each strategy is comprised of up to six heuristic algorithms to form two offensive strategies and two defensive strategies. The two search algorithms are minimax and alphabeta and are used to search the search tree. These are all described in the **Algorithms** section below.

The game runs six "matches", all players use alphabeta and play on an 8 by 8 chessboard unless noted otherwise:

1. minimax offensive 1 VS alphabeta offensive 1
2. offensive 2 VS defensive 1
3. defensive 2 VS offensive 1
4. offensive 2 VS offensive 1
5. defensive 2 VS defensive 1
6. offensive 2 VS defensive 2
7. offensive 2 VS defensive 2 on a 5 by 10 chessboard

File Structure

The following files should be included in the same folder as this document:

- WhitePawn.png
    - Image used in display.pi
- BlackPawn.png
    - Image used in display.pi
- Null.png

    - Image used in display.pi
- README.txt
    - Provides instructions on how to run the program.
- Breakthrough.py
    - The “main” script, run this file to show the AI playing breakthrough in the required matchups.
- player.py
    - Defines the player class and the heuristic functions used in the players’ strategies.
- boardV2.py
    - Defines the board class and the functions required for the act of playing breakthrough, such as functions to run a game or select moves.
- brainV2.py
    - Defines the getPossibleStates function to build a search tree of possible moves, and defines the minimax and alphabeta functions.
- treeV2.py
    - Defines the node class use to build search trees.
- display.py
    - Defines the functions that implement the GUI display.

In order to run the game, make sure you have python Version 3 installed (https://www.python.org/downloads/) and that you have pygame installed (https://www.pygame.org/download.shtml). In Windows, simply double click Breakthrough.py. Alternatively, or if double clicking the file does not work) you may open the folder in Visual Studio and run Breakthrough.py with or without the debugger. Running without the debugger is recommended as the debugger can greatly slow down computational time.

Problem Description

The process of representing a chessboard in computer memory and moving its pieces around within a set of parameters defined by the game’s rules are rather straightforward. Even implementing the action of two players alternating between each other and moving pieces in a turnwise fashion is programmatically straightforward.

The problem of this game comes from *choosing* which move to make on any given turn. From one possible game state you may reach a wide variety of other game states by moving just one piece. Even on just the first turn of the game the first player has to pick from 22 possible and otherwise equivalent moves.

This is exasperated by the problem of quantifying what it means to be a good move, and how to compare multiple available moves and select the better one, not to mention how to choose between heuristically equivalent moves. Also, a good AI will avoid moves that result in losing the game and make moves that result in it winning the game.

Additionally, the computational time required to select one move of many in a game with as many possible moves as Breakthrough can be quite dramatic depending on system resources. A good AI will implement methods that cut down on its search time.

The full solution to these problems is described in the **Algorithms** section below, but in short, for each turn, the player builds a tree of all possible moves available from the current state of the board. This tree is three "*layers*" deep, taking into consideration its opponent's next moves as well. It scores each node of the tree based off some strategy, which is in turn based off some heuristic scores. It selects one move to make based off of these heuristics.

Algorithms

Get Possible States

In `brainV2.py` see the `getPossibleStates` function.

This function takes in the current state of the board and builds a search tree of nodes. Each node contains a list of all possible states which can be reached by the current player, starting with the current state as the root node. The tree is built to be three layers deep: the root node or current state of the board being the 0th layer, the 1st layer being all states reachable from the current state with one move by the current player, the 2nd layer being all the states reachable from all those states with a move from the current player's opponent, etc. to the 3rd layer.

On the third layer, each node is also assigned a heuristic score that indicates how desirable that state is according to the player's strategy. The strategy is a list of heuristic functions, each of which is evaluated based on the board state contained in the node. getPossibleStates then calls a high heuristic function, which takes the sum of all heuristics in the player's strategy and assigns this value to the heuristic of the node.

Minimax

In `brainV2.py` see the `minimax` function.

This function is applied to a tree of nodes, such as the one created with getPossibleStates. It iterates to the second layer of the tree, the parents to the terminal nodes, and for each of those nodes, searches its children (the terminas) for the node with the highest heuristic value. This is then recursively pushed up the tree, to the first layer. This first layer is then what is used later for the player to select their optimal move.

Alphabeta

In `brainV2.py` see the `alphabeta` function.

This function, like minimax, is applied to a tree of nodes. Alphabeta is improvement upon the Minimax algorithm. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It begins by searching the leftmost branch and searches the terminal nodes with the highest heuristic value and stores that number when looking at the next branch. When it finds a highest hurestic with a larger number in the first terminal node the algorithm breaks from the loop and does not search the rest of the branch. This improvement on Minimax cuts down on computational time by reducing the number of times comparison happen.

Heuristics

See `player.py` for these functions.

Player Heuristics

There are several heuristic functions available to the player class (see **The Player Class** section, below). Each player is given a strategy characteristic, that comprises of a list of heuristics. In this rendition of the game, 6 were designed, but in theory the software can handle any new heuristics that can be designed just by adding them to the player object's strategy.

`offensiveHeuristic`: This heuristic is designed to guide the player to remove the opponent's pieces from the board. It assigns a higher score to board states which have fewer of the opponents pieces on the board compared to the amount of pieces each player starts with. A final score is returned from this function as a fraction between 0 and 1 (0 indicating no pieces have been removed, 1 indicating all opponent pieces have been removed) multiplied by a weight. So if on an 8 by 8 board, where each player starts with 16 pieces, and 5 of the opponent's pieces have been removed, the state of the board would return an offensive heuristic of (16 - (16 - 5) * w) / 16 = 5w / 16 = 0.3125w, w being the weight. The weight selected for this algorithm was 3, and indicates how preferable this strategy is compared to other strategies: 3 was selected because it was a small number that through experimentation showed to AI to act best according to its strategy without drowning out other heuristics. What this means it that the heuristic returned a number between 0 and 3, with higher heuristics being more optimal.

`defensiveHeuristic`: This heuristic is designed to guide the player to maintain its number of pieces on the board. It assigns a higher score to board states which have more of its pieces compared to how many it started with. This functions similarly to offensiveHeuristic in that the score is returned as a ratio of the number of pieces on the board and the starting pieces, except that it counts up for each piece instead of the offensive heuristic which counts down for each piece. So, if on an 8 by 8 board, where each player starts with 16 pieces, and the player has 12 of them left, it would return (12/16) * w = .75w, w being a weight as described above. For this implementation a weight of 3 was also selected for the same reasons as above, so the heuristic also returns a final score between 0 and 3, 0 indicating the player has no pieces on the board and 3 indicating all of its pieces remain.

`runForward`: This heuristic, intended to serve the role of “offensive heuristic 2”, is designed to overcome defensiveHeuristic. The strategy selected by the programmers was to move forward as quickly as possible and break the opponent’s defenses, and trials have shown that strategies with runForward tend to beat strategies that focus on defensiveHeuristic. This function works by finding the player’s piece that is closest to the opponent’s home row, and subtracting its distance from the home row from the total number of rows. This will tend to favor board states in which the player has even one piece that is as far along as possible, and tends to influence the AI to select board states that move that piece forward. The score is calculated by finding the number of rows on the board, starting from the home row of the opponent, and subtracting one for each row until the algorithm finds a piece of the player, then dividing that by the number of rows and multiplying the result by a weight. The weight chosen for this implementation was 1, so thusly has no effect, and as such this function returns a score between 0 and 1.

`moveWall`: This heuristic, intended to serve the role of “defensive heuristic 2”, is designed to overcome offensiveHeuristic. The strategy selected was to move the players pieces along sequentially, side by side, so that a wall or row of pieces gets moved forward together in unison. While not as successful at defeating offensiveHeuristic as runForward was at defeating defensiveHeuristic, it showed some success and was retained, although more optimal heuristics are certainly possible. It is found by finding the most progressed piece of the player, similar to runForward, then counting the total number of pieces in that row, and dividing it by the number of columns. This gives a higher score to states where the row with the most forward piece has more pieces in the same row. As before, this score can be weighted, but a weight of 1 was selected for this implementation, and the function returns a score between 0 and 1.

`aboutToWin`: This heuristic was designed to allow players to understand the win condition of the game and strive for that state. This gives a flat weighted score to states where the player has won the game by moving a piece to the opponent’s home row. The weight was selected to be 5 in this implementation to strongly influence player’s to take that winning move. This function was added as it was observed that without some kind of win condition heuristic, player’s would often be one move from winning but would instead move other pieces that gave higher heuristics based off those strategies.

`aboutToLose`: This heuristic was designed to give a lower score to board states in which the opponent has won. It functions similar to aboutToWin, in that is simply assigns a flat score to a board state, also weighted at 5, but returns a negative value. So board states in which the opponent has won receive a -5 penalty to their heuristic score for players who use this function as part of their strategy.

High Heuristic

This heuristic (`highHeuristic`) instead of just taking a single board state metric into consideration, takes into account a player's entire strategy. It iterates through every function in the player strategy (like those described above) and returns their sum. Since the scores are weighted in the individual heuristics, this function can return an unweighted sum.

Although, in order to break ties between heuristically optimal board states, this function also has the option to add a random number to the final heuristic. This option was implemented in this implementation of the program. Furthermore, this random noise was divided by 100, to ensure that it would not interfere with any of the other heuristic scores and is only used as a tiebreaker between board states that would have been the same. Without this, the first several moves of the game for each player (with the singular exception of players with runForward in their strategy) would have been identical and would have given an unfair bias in the effectiveness of the other heuristics once they came into play. This randomness ensures a new game is played every time without interfering with the player strategy.

It is worth mentioning also that this final heuristic is the one score that gets assigned to a node when building the player's search tree for that turn. Also, negative high heuristics tended to break selection algorithms in other parts of the program, so negative scores are simply reset to 0 here. It was also discovered that players using the runForward and moveWall heuristics benefitted from also having offensiveHeuristic and defensiveHeuristic respectively in their strategy as well, and acted more intelligently when their strategies were more inclusive. This is also why aboutToWin and aboutToLose were designed, so games did not drag on or end prematurely when obvious moves to win or avoid loss could be taken.

Implementation

The program has two versions, a first version to get the basic idea of the program functioning, which only allowed for one offensive player to play against one defensive player on a static board of one size; and a final, second version, which allowed for any number of player objects and board objects to be instantiated with their own characteristics and any number of games to be run. This second version was based around a player class and a board class.

The Board Class

See `board` in `boardV2.py`.

The board class is designed to create a board object. A board contains a two dimensional array called the field that may be of any width and height, that contains either blank spaces or player tokens. The field contains a list of strings, with the string representing a blank space represented in the object. The number or rows and the number of columns is also stored in the board object.

The board also contains a list of valid moves each represented by a single character. As of right now this version of the software is only designed to handle on possible list of moves: `['L', 'F', 'R']`, representing forward, left diagonal, and right diagonal, although the class was designed to hold any list so if certain other functions are updated to accommodate more or less moves the class would not have to be updated.

It is also worth mentioning that board.py is where several other important functions are stored:

- `setStartingPieces` takes in two players and a board and places the first player's token in the last two rows of the board and the second player's token in the first two rows of the board, overriding the blank spaces on the board's field and orienting the board as it would be viewed from above by the first player.
- `isOnBoard` checks to ensure given coordinates are on the given board and is used in makeMove to determine if moves are valid.
- `makeMove` checks to see if a move is valid and if it is, makes that move on the board's field. It returns True if the move was successful or False if it was invalid.
- `findCapturedPieces` compares two board states against a player to see if that player has captured any of its opponent's pieces. Since it compares a board state after a single move, only one piece could be captured at a time, so it simply returns True or False.
- `bestMoveIndex` finds the best move a player could make in the first layer of their search tree. It takes in the tree's root and searches all possible moves reachable from that point and return the one with the highest heuristic. Also, it will not consider states where the opponent could win the game in its next turn and will always take a winning move. It returns the index in the node's next turns list where this optimal game state is stored. This is used only after a search algorithm like minimax or alphabeta has been applied to the tree.
- `runGame` plays a game between two given players on one given board. It is described in more detail below under **Running the Game**.

The Player Class

See `player` in `player.py`.

The player class is designed to create a player object. A player contains a string to represent its token on the board, a turn order (either 0 for the first player or 1 for the second player), a heuristic indicator, 1 or -1, to designate whether the player is searching for high or low heuristics respectively (all players in this implementation use high heuristics), an algorithm indicator (a boolean) to determine whether they use alphabeta (True) or just minimax (False), a board that they are designated to play the game on, an opponent player object, and finally strategy.

The player strategy is comprised of a list of heuristic functions. It is stored so that any of these heuristics may be called by utilizing the reference to the player object, but it is mainly used in the highHeuristic function described above.

Two players are required to play a game, and player.py also includes functions to set two players as opponents against each other by storing each other in each player's opponent component. player.py is also where all heuristics are defined (see the **Heuristics** section above).

Running the Game

See `Breakthrough.py` and `runGame` in `boardV2.py`.

Everything described up to this point has been designed to build a framework in order to execute a digital and graphical (see **Displaying the Game** below) representation of the game of Breakthrough. Many functions and class definitions have been described, but the actual game is executed in the file Breakthrough.py file.

Breakthrough.py initializes the possible legal moves, `['L', 'F', 'R']`, and defines four strategies as lists of player heuristics, two offensive and one defensive. All heuristics consist of aboutToWin and aboutToLose, the defensive ones include defensiveHeuristic, and the offensive ones include offensiveHeuristic. One defensive strategy additionally includes moveWall, and one offensive strategy additionally includes runForward.

Once these references are established, the program builds players and boards for seven games. The first six games are played on an 8 by 8 board while the last is played on a 5 by 10 board, but the board class could also be used to create boards of other sizes, but note that boards less than 2 rows or 2 columns would break other algorithms.

The players for each game are built to play the following matchups:

1. Minimax Offensive Heuristic 1 VS AlphaBeta Offensive Heuristic 1
2. AlphaBeta Offensive Heuristic 2 VS AlphaBeta Defensive Heuristic 1
3. AlphaBeta Defensive Heuristic 2 VS AlphaBeta Offensive Heuristic 1
4. AlphaBeta Offensive Heuristic 2 VS AlphaBeta Offensive Heuristic 1
5. AlphaBeta Defensive Heuristic 2 VS AlphaBeta Defensive Heuristic 1
6. AlphaBeta Offensive Heuristic 2 VS AlphaBeta Defensive Heuristic 2
7. AlphaBeta Offensive Heuristic 2 VS AlphaBeta Defensive Heuristic 2 (oblong board)

Once the board and player objects are built, for each matchup, the `runGame` function from `board.py` is called. runGame sets each player's starting pieces, and initializes a list of game statistics, these lists are used to report on various ratings of the matchup including:

- the winner of the game
- the number of turns played
- the final state of the board (the 'field' variable of the board object)
- the total nodes expanded by each player's search tree
- the total pieces (workers) captured by each player
- the average runtime for each of the two search algorithms (minimax or alphabeta)

runGame runs a while loop dependent on whether the current player's (beginning with the player with a 0 turn indicator) opponent has not won the game (this will always be true after starting pieces have been set on a blank board). This loop repeats turns by taking the current field of the board object and building a search tree using getPossibleStates of all possible moves three layers deep, assigning heuristics to the terminal nodes based off the current player's strategy, with the current field as a root node. The heuristics are then searched based off the current player's algorithm, either minimax or alphabeta.

Also, the node counters and algorithm timers are incremented as appropriate.

After this, the heuristically best possible move to make is selected using the bestMoveIndex function as described above (see **The Board Class** section above), while giving priority to winning moves or moves that take opponent's pieces that are about to win.

If the current player captured pieces, these counters are updated.

Finally, the turn counter is incremented and the current player is set to the current player's opponent. This while loop continues until the game has been won by a player. Afterwords, the game statistics are updated and returned by runGame.

Displaying the Game

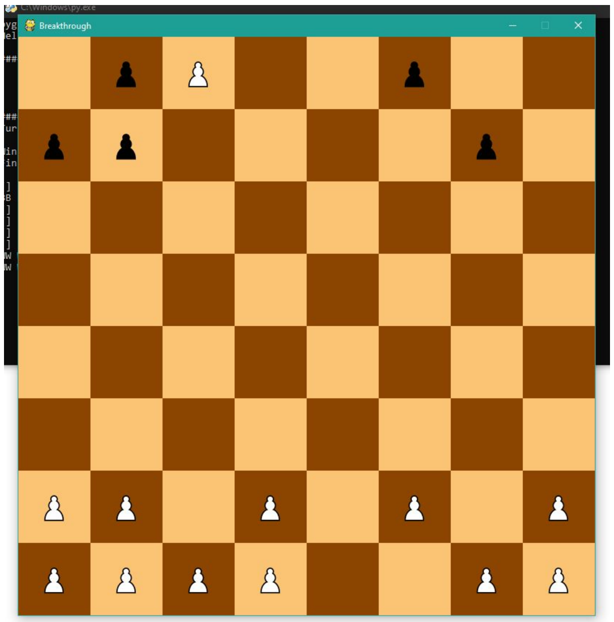
See `display.py` and `runGame` in `boardV2.py`

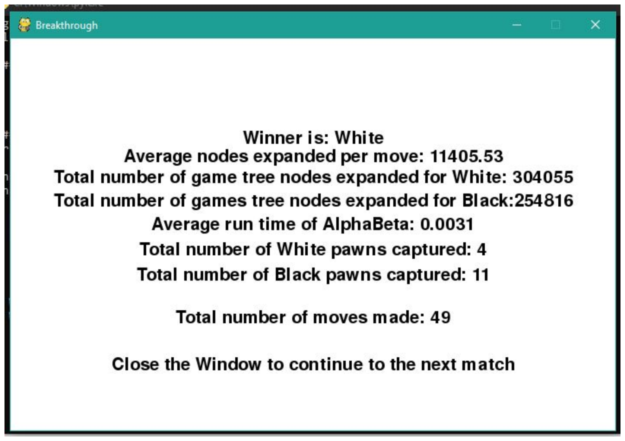
In order to run this in an IDE Pygame must be installed. This program uses pygame which is an open source python programming language library for making multimedia applications like games. The main function in this file is the `draw_board` function. It takes in the current state of the board from runGame and sets the size of the screen in pixels to the size of the row and column of the array, times a number to make the screen bigger and then fill it with alternating colors of light brown and dark brown to create a checkerboard. Then the function searches the array for everytime that there is a "BB" or "WW" token on the board and places an image of a white pawn or black pawn respectively in the center of the square. This is done after every turn is made.

Run Results and Analysis

Listed below are the results of the seven matchups described above:

1. Minimax Offensive Heuristic 1 VS AlphaBeta Offensive Heuristic 1



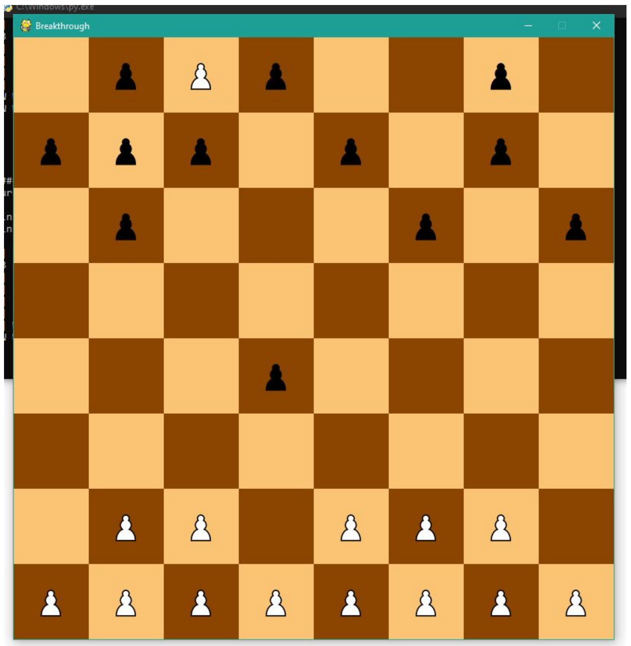


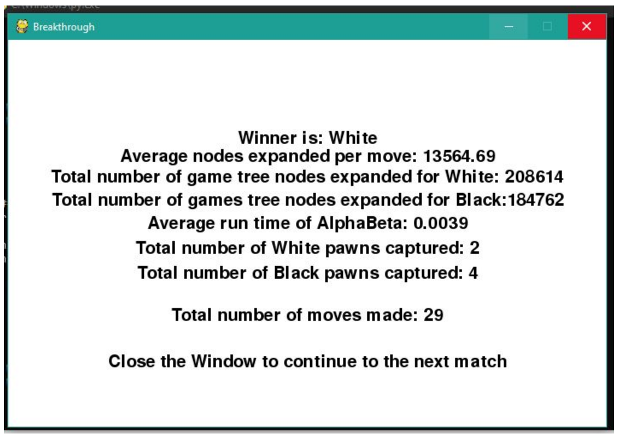
```
Winner is: White
Average nodes expanded per move: 11405.53
Total number of game tree nodes expanded for White: 304055
Total number of games tree nodes expanded for Black:254816
Average run time of AlphaBeta: 0.0031
Total number of White pawns captured: 4
Total number of Black pawns captured: 11

Total number of moves made: 49

Close the Window to continue to the next match
```

2. AlphaBeta Offensive Heuristic 2 VS AlphaBeta Defensive Heuristic 1



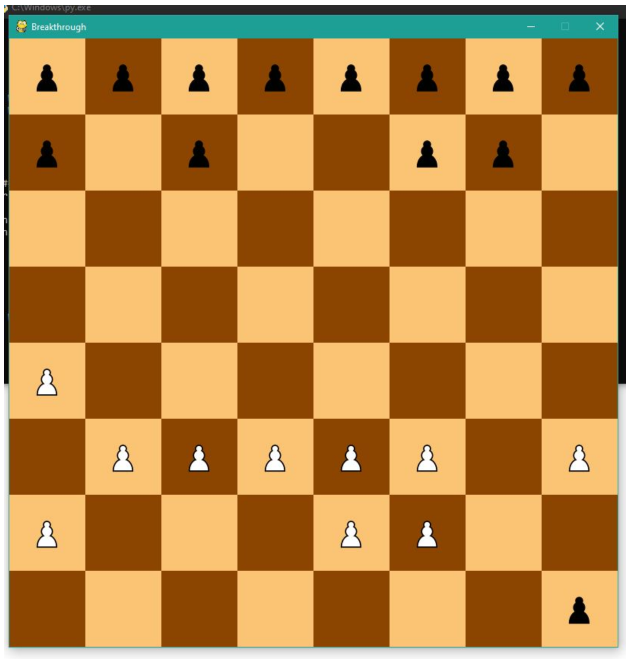


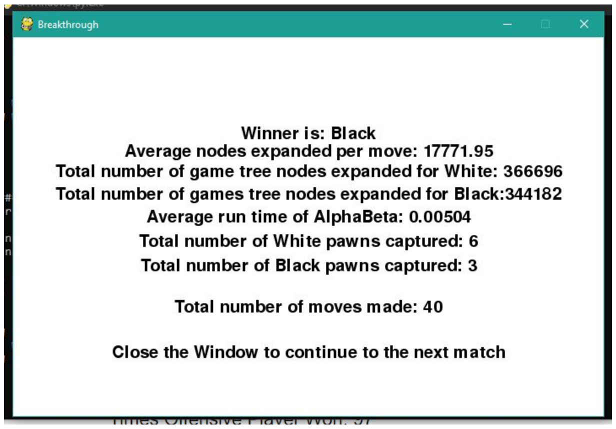
```
Winner is: White
Average nodes expanded per move: 13564.69
Total number of game tree nodes expanded for White: 208614
Total number of games tree nodes expanded for Black:184762
Average run time of AlphaBeta: 0.0039
Total number of White pawns captured: 2
Total number of Black pawns captured: 4

Total number of moves made: 29

Close the Window to continue to the next match
```

3. AlphaBeta Defensive Heuristic 2 VS AlphaBeta Offensive Heuristic 1



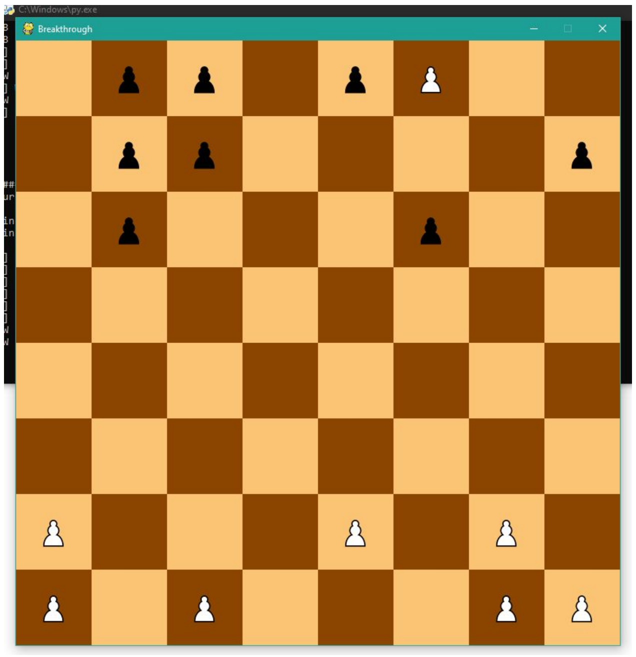


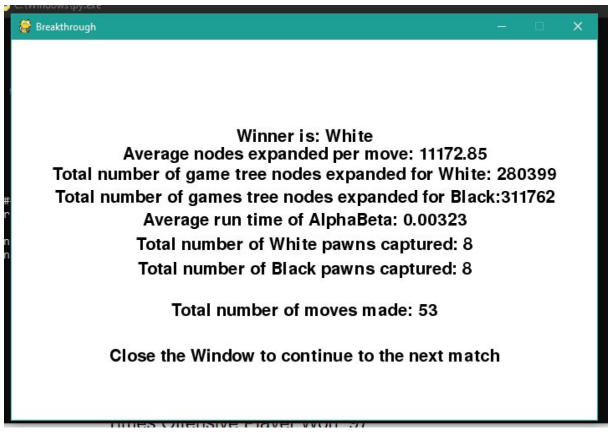
Winner is: Black
Average nodes expanded per move: 17771.95
Total number of game tree nodes expanded for White: 366696
Total number of games tree nodes expanded for Black:344182
Average run time of AlphaBeta: 0.00504
Total number of White pawns captured: 6
Total number of Black pawns captured: 3

Total number of moves made: 40

Close the Window to continue to the next match

4. AlphaBeta Offensive Heuristic 2 VS AlphaBeta Offensive Heuristic 1



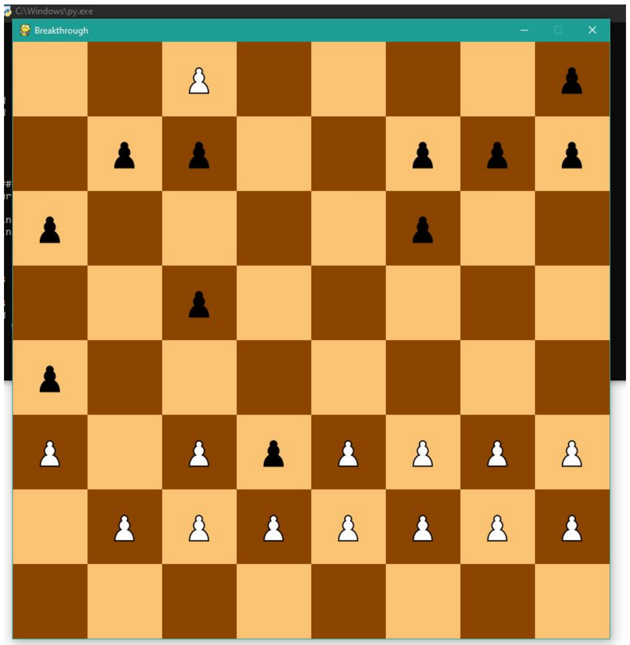


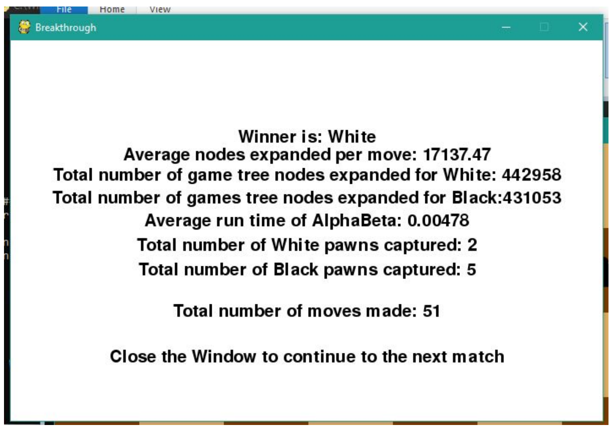
Winner is: White
Average nodes expanded per move: 11172.85
Total number of game tree nodes expanded for White: 280399
Total number of games tree nodes expanded for Black:311762
Average run time of AlphaBeta: 0.00323
Total number of White pawns captured: 8
Total number of Black pawns captured: 8

Total number of moves made: 53

Close the Window to continue to the next match

5. AlphaBeta Defensive Heuristic 2 VS AlphaBeta Defensive Heuristic 1



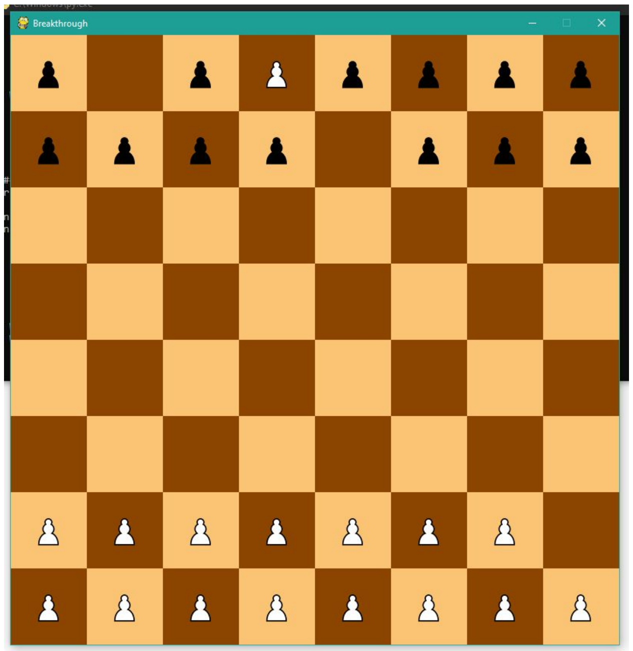


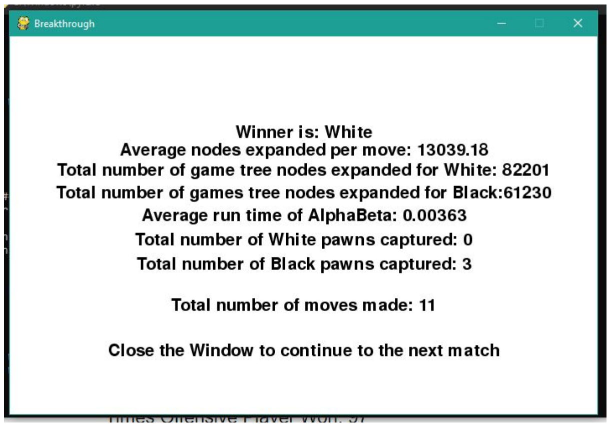
```
Winner is: White
Average nodes expanded per move: 17137.47
Total number of game tree nodes expanded for White: 442958
Total number of games tree nodes expanded for Black:431053
Average run time of AlphaBeta: 0.00478
Total number of White pawns captured: 2
Total number of Black pawns captured: 5

Total number of moves made: 51

Close the Window to continue to the next match
```

6. AlphaBeta Offensive Heuristic 2 VS AlphaBeta Defensive Heuristic 2



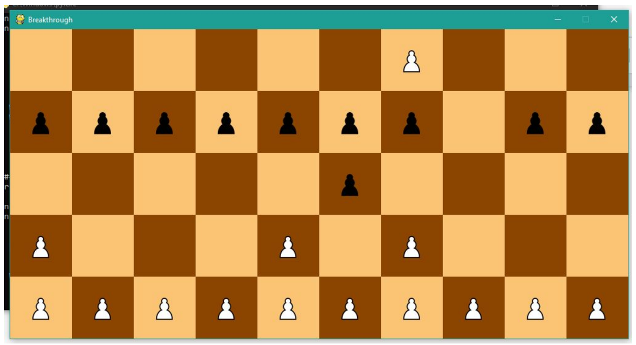


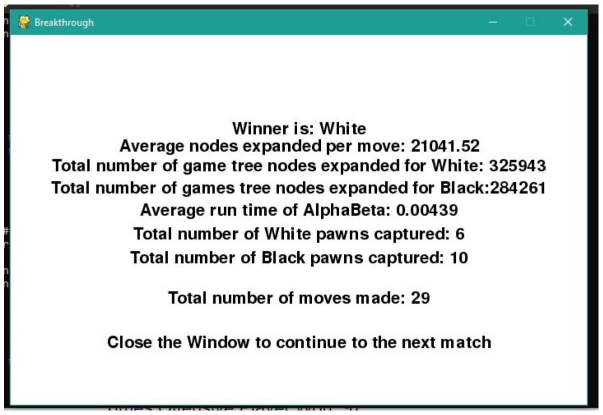
```
Winner is: White
Average nodes expanded per move: 13039.18
Total number of game tree nodes expanded for White: 82201
Total number of games tree nodes expanded for Black:61230
Average run time of AlphaBeta: 0.00363
Total number of White pawns captured: 0
Total number of Black pawns captured: 3

Total number of moves made: 11

Close the Window to continue to the next match
```

7. AlphaBeta Offensive Heuristic 2 VS AlphaBeta Defensive Heuristic 2 (oblong board)





Winner is: White
Average nodes expanded per move: 21041.52
Total number of game tree nodes expanded for White: 325943
Total number of games tree nodes expanded for Black:284261
Average run time of AlphaBeta: 0.00439
Total number of White pawns captured: 6
Total number of Black pawns captured: 10

Total number of moves made: 29

Close the Window to continue to the next match

An additional file (see `analysis.py`) was designed, coded, and implemented to run a large number of matchups and taking the average of a number of statistics they generated. This implementation ran 99 games each of four matchups between an offensive and a defensive player, and finally an additional round of 99 games for a matchup between two players each using all implemented heuristics. (The total runtime of this script took about 25 hours.) The results are shown below:

```
Reporting data for Offensive 1 vs Defensive 1
Games analyzed: 99
Average Number of Turns: 41.121212121212125
Average Offensive Player Nodes Expanded: 318348.35353535356
Average Defensive Player Nodes Expanded: 275856.22222222225
Average Offensive Player Workers Captured: 2.3535353535353534
Average Defensive Player Workers Captured: 7.040404040404041
Times Offensive Player Won: 97
Times Defensive Player Won: 2

Reporting data for Offensive 2 vs Defensive 1
Games analyzed: 99
Average Number of Turns: 35.85858585858586
Average Offensive Player Nodes Expanded: 257314.0707070707
Average Defensive Player Nodes Expanded: 232355.81818181818
Average Offensive Player Workers Captured: 2.6363636363636362
Average Defensive Player Workers Captured: 6.555555555555555
Times Offensive Player Won: 98
Times Defensive Player Won: 1

Reporting data for Offensive 1 vs Defensive 2
Games analyzed: 99
Average Number of Turns: 36.61616161616162
Average Offensive Player Nodes Expanded: 282836.68686868687
Average Defensive Player Nodes Expanded: 254924.56565656565
Average Offensive Player Workers Captured: 2.272727272727273
Average Defensive Player Workers Captured: 6.878787878787879
Times Offensive Player Won: 99
Times Defensive Player Won: 0

Reporting data for Offensive 2 vs Defensive 2
Games analyzed: 99
Average Number of Turns: 36.5959595959596
Average Offensive Player Nodes Expanded: 263049.43434343435
Average Defensive Player Nodes Expanded: 232727.36363636365
Average Offensive Player Workers Captured: 2.6363636363636362
Average Defensive Player Workers Captured: 7.01010101010101
Times Offensive Player Won: 99
Times Defensive Player Won: 0

Reporting data for All Heuristics for both players
Games analyzed: 99
Average Number of Turns: 45.18181818181818
Average Offensive Player Nodes Expanded: 279409.89898989897
Average Defensive Player Nodes Expanded: 249937.9393939394
Average Offensive Player Workers Captured: 5.626262626262626
Average Defensive Player Workers Captured: 6.787878787878788
Times Offensive Player Won: 51
Times Defensive Player Won: 48
```

Conclusions

The first and most obvious conclusion to make based off these results is the dominance of offensive heuristics over defensive heuristics. This makes sense, as defensive heuristics have minimal interest in capturing other opponent's pieces, and capturing a piece tends to move one's piece closer to the opponent's home row. Other than capturing a piece that's about to capture one of your pieces, all possible moves are defensively equivalent since they all have the same number of your pieces on the board. As a result, defensive players tend to move more randomly, while offensive players tend to be more progressive.

This is exasperated by the implementation of Offensive 2 and Defensive 2, as Offensive 2 specifically runs forward until you have the opportunity to capture a piece and Defensive 2 only allows you to progress forward if your most progressed row has as many pieces as possible in it. This speeds up the progress of Offensive 2 and slows the forward momentum of Defensive 2.

Also, an interesting finding was that the implementation of Offensive 2 and Defensive 2 reduced the average number of turns per game by over ten turns. However, when both players used all strategies the average amount of turns shot back up. It is possible that fully optimized players were more competitive, and thus it took longer to finish a game. While without Offensive 2 or Defensive 2 players could not plan as well and were more lethargic or spent more time moving randomly until they could interact with each other by capturing pieces. Offensive 2 and Defensive 2 added patterns to recognize that moved forward, while Offensive 1 and Defensive 1 are always equivalent if the players are too far away from each other to capture pieces.

An interesting note is that defensive players tend to capture more pieces than offensive players, despite offensive players winning more games. This is because offensive players are typically more efficient, once they break through the defensive player's defenses, they can simply run forward to the end. They are focused on capturing pieces, and that pushes them forward, and in going forward offensive players seem to tend to "sacrifice" more of their pieces. The defender is successful in maintaining their numbers, but not in moving forward to take the win.

This does not necessarily mean that in a real world game of Breakthrough, playing "offensively" is better than playing "defensively". It just means that an offensive heuristic as defined by this implementation of this program tends to move pieces to the end game state faster than defensive ones.

Statement of Contribution

Christopher Lebovitz

- Supervised team meetings and project orientation.
- Established and implemented version control hierarchy (github) and file structure.
- Designed and coded the original board item.
- Designed and coded the original node class.
- Designed and coded the Alphabeta algorithm.
- Designed, coded, and implemented the GUI and display of turn by turn graphic board and game results.

Dylan E. Wheeler

- Redesigned node class for Version 2.
- Designed and coded new board class.
- Designed, coded, and implemented player class.
- Designed and coded heuristics and player strategies.
- Designed and coded search tree building.
- Designed and coded functions to implement breakthrough game rules and run the game.
- Designed and coded the Minimax algorithm.

- Coded and implemented performance analysis.

Advanced Exploration Submitted for Bonus Points

In addition to the standard requirements for the assignments, the following design aspects were implemented, each is described in more detail in this report above:

- Implementation of a GUI to display games being played turn by turn.
- Addition of heuristics and decision making that take the win condition into account, including the win condition of the opponent. The AI will move toward winning positions, take winning moves, and prevent the opponent from winning if possible.
- Designed the program to handle a board of any dimensions, played an additional game on an oblong 5 by 10 board.
- Programmed an additional series of 5 matchups to run 99 games of each (for a total runtime of 25 hours) in order to compare heuristics for analysis. See `analysis.py`.