CM1 Abstract Games for Two Players

An Analysis of Heuristics in Reversi

By

Michael Roddy,

Arran Tyson &

Shane Dowd

Project Supervisor

Conn Mulvihill

16/04/18

Prepared for

CT328 – Final Year Project

Abstract

Game playing, as one of the most challenging fields of artificial intelligence has received a lot of attention. Games like Reversi/Othello, which have proven to fit in well with computer game playing strategies, have spawned a lot of research in this direction.

Though numerous computer Reversi players have been designed, and have beaten human world champions, it is not very clear as to how the various Reversi heuristics interact. This paper implements and examines various heuristics, in an attempt to make observations about the interplay between the heuristics and how well each heuristic contributes as a whole. The focus of our project is to explore heuristics that apply to the abstract game Reversi/Othello. We chose Reversi as the main focus for our project because the Reversi board game makes for a good environment for tractably exploring heuristics as the complexity of the game is not too great, nor too simple. For example, Reversi has a branching factor of 10, which is more complex than Tic-tac-toe which has a branching factor of 4, but not as complex as chess which has a branching factor of 35.

Our project will look at deconstructing, testing, ranking and analysing existing heuristic elements in existing work (heuristics meaning strategies) developed by Sweigert. Then, we will look at developing new combinations of heuristic elements from this work and also developing an entirely new heuristic.

1. Introduction

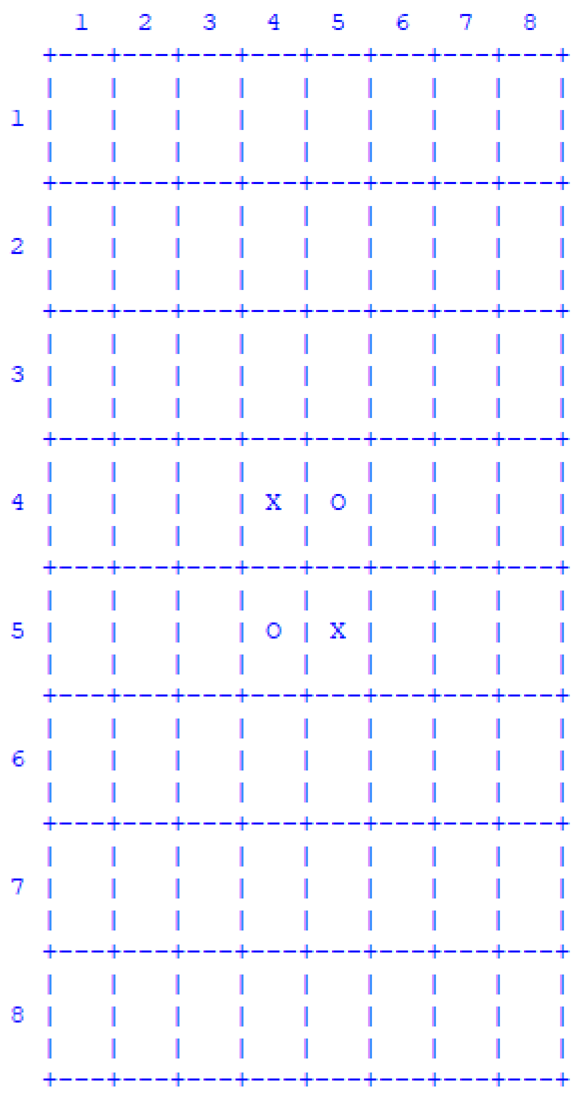
The title of our final year project is "Abstract game for two players". For example, an abstract game for two players may include Connect Four, Tic-tac-toe or Nim. Game playing, as one of the well-admired components of artificial intelligence research, has captured tremendous amounts of attention. Computers, looking ahead beyond the next move and judiciously rationing out the next best move, mimic human intelligence, and at times, surpass it. Computers, to a decent extent, have captured the essence of game playing along with its intricate complexities. The importance of game playing arises out of the competition that exists between the human race and machines. It's a race to prove superior intelligence. Dominant Reversi-playing computer programs play very strong against human opponents. This is mainly due to difficulties in human "look-ahead" abilities, meaning how far a human can "look" and evaluate different moves and positions in the future. Therefore, Reversi-playing computer programs have easily defeated humans since 1980. A Reversi-playing computer program named "The Moor" beat the reigning world champion at the time. One of the most popular two person board games of all time is of course chess and in 1996 Garry Kasparov, who was at the time, ranked the number one chess player in the world, was beaten by the chess-playing computer program named "Deep Blue" which was developed by IBM. This victory for machines set the scene for future board game-playing computer programs such as Deep Thought, Watson and Alpha Go.

Reversi, a board game derived from Go, has been an example where computers have exemplified great game play, beating human world champions. The primary reason being the small branching factor of 10, allowing the computer to look ahead in abundance, thrashing human intuition and reasoning. As processors increase in speed and complexity, the ability of computers to reason beyond the current state increases, while human intelligence maintains a fairly static average performance over generations. This leads to an extending gap in the Reversi-playing ability of humans and computers. The massive success of Reversi, apart from the small branching factors involved, can also be attributed to heuristic functions successfully representing the state of the game. Heuristics in Reversi, suffer from few pitfalls, when chosen cautiously. The typical calculated heuristic value is a linear function of various different heuristics. We implement, analyse and test various heuristics and strategic elements to determine the contribution of each heuristic to the entire game play. In the process, we also determine the manner in which each heuristic interacts with the others. Understanding the role of each heuristic in game play would enable us to create our own new and improved heuristic or strategy.

In section 2, we offer a description of the rules of Reversi for the reader. In section 3, we discuss technologies used and document how the system works. In section 4, we will examine existing heuristics/strategies developed by Sweigert and test them to see how well they perform. Section 4 will also present experimental results/analysis of our tests. Section 5 will show the development of our own heuristic element and also the testing and results/analysis of its performance against Sweigerts original heuristics. We will conclude with an evaluation and summary of our work in section 6.

2. Reversi Rules

Description
Reversi belongs to the family of board games that were derived from Go. The game consists of an 8x8 grid with 64 squares in total. We assign each column from left to right with a number starting from 1 and each row is also numbered from top to bottom starting with 1. This is the notation that shall be followed throughout the rest of the paper. The initial configuration of the board is shown in Figure 1. Each player is associated with a letter, either X or O. A player owns the squares in which his/her letter is placed. O initially owns (5,4) and (4,5) while X initially owns (5,5) and (4,4).



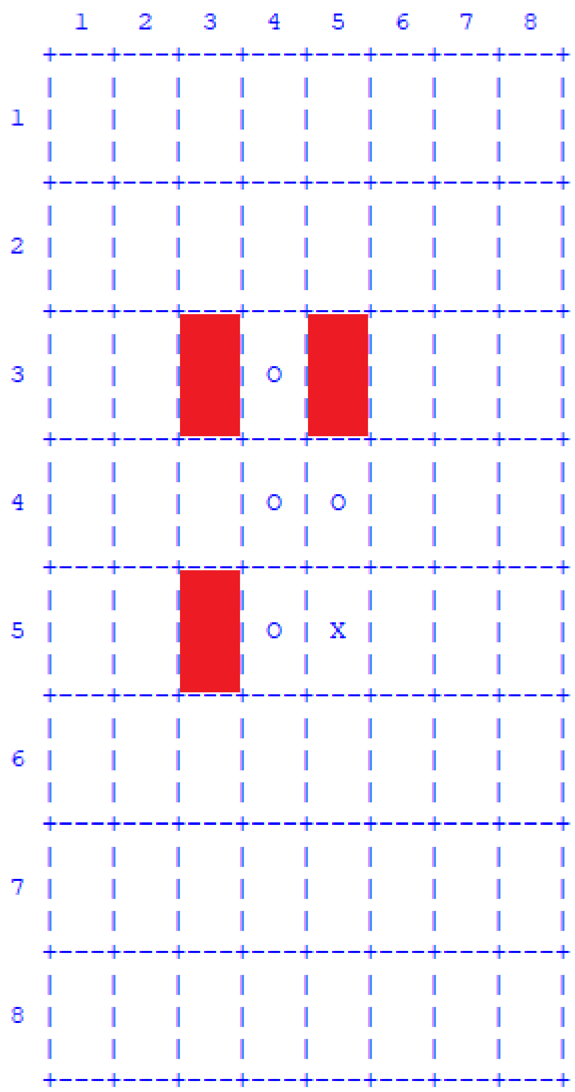
```
     1   2   3   4   5   6   7   8
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
 1 |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
 2 |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
 3 |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
 4 |   |   |   | X | O |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
 5 |   |   |   | O | X |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
 6 |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
 7 |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
   |   |   |   |   |   |   |   |   |
 8 |   |   |   |   |   |   |   |   |
   |   |   |   |   |   |   |   |   |
   +---+---+---+---+---+---+---+---+
```

Figure 1: Showing the starting game configuration for Reversi

Objective
The objective of Reversi is to finish the game with more "pieces" or letters on the board in his/her own alphabetical denomination than his/her opponent.

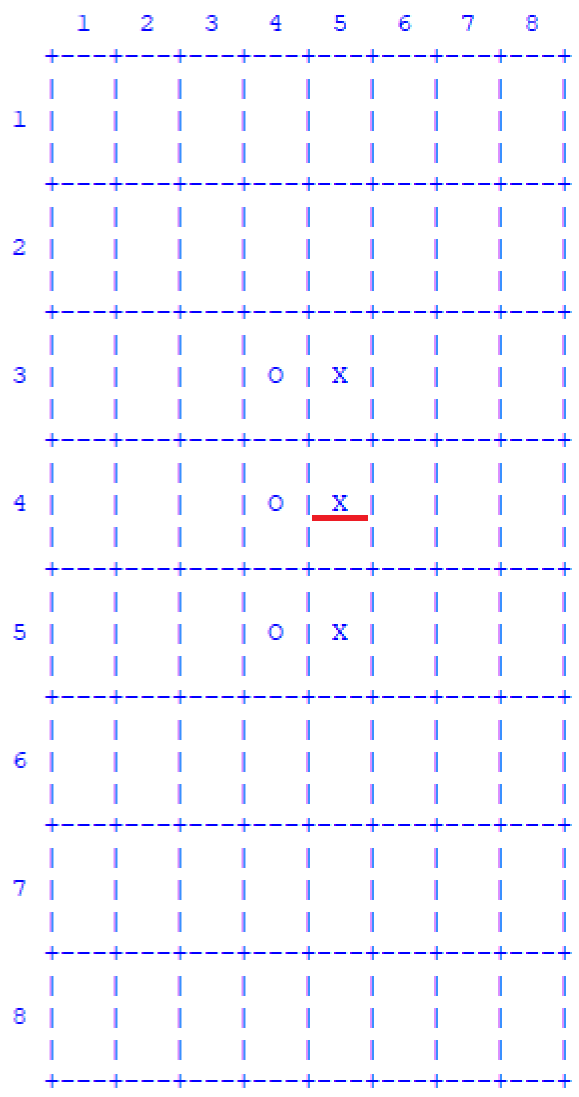
Start of game
Before starting, players will decide which letter each will use. Next, four letters are placed in the central squares of the board so that each pair of letters of the same Alphabetical denomination form a diagonal between them. X always moves first and only one move is made every turn.

Moves
The game progresses as each player makes moves. A move is made by placing a letter in an empty square. When a player does so, all of the letters bracketed between the newly placed letter and another letter of the same denomination get *flanked* into the denomination of the newly placed letter. A move must *flank* at least one or more of the opponent's letters. If it is not possible to make a move, the persons turn is forfeited and the opponent makes another move.
The *flanking* of letters is shown in Figures 2a and 2b.



```
      1   2   3   4   5   6   7   8
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  1 |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  2 |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  3 |   |   |   | O |   |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  4 |   |   |   | O | O |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  5 |   |   |   | O | X |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  6 |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  7 |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
    |   |   |   |   |   |   |   |   |
  8 |   |   |   |   |   |   |   |   |
    |   |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+---+
```

Figure 2a: The highlighted red squares show valid moves for X



```
    1   2   3   4   5   6   7   8
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
1 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
2 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
3 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
4 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
5 |   |   |   | O | X |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
6 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
7 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |   |   |
8 |   |   |   |   |   |   |   |   |
  |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
```

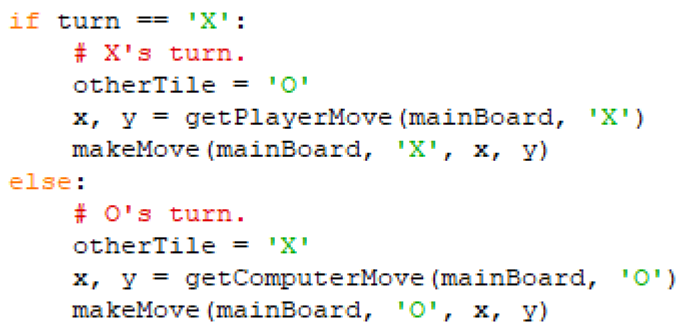
Figure 2b: The underlined letter shows where O was flanked by X's placing of his/her letter in (5,3)

End game
The game ends when all of the squares on the board are taken or none of the players can make a move. In any case, the winner is the player who has more letters on the board. The game ends in a draw when both players have the same number of letters on the board.

3. Implementation

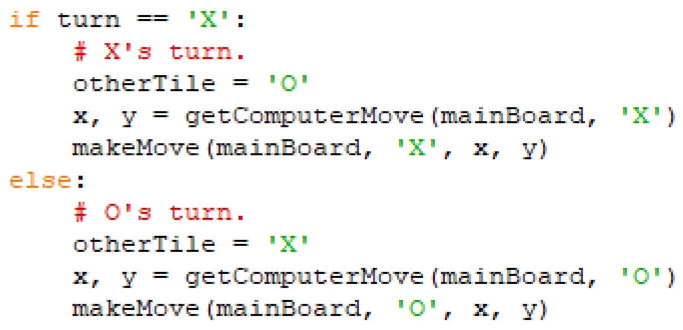
Our project implemented the Reversi game along with various heuristics in the Python Integrated Development Environment. The entire code base consisted of over 300 lines of code. The technologies we used were developed by Al Sweigert in his book "Invent With Python". The original Reversi program *AISim1.py* from chapter 15 had two functions, *getPlayerMove()* and *getComputerMove()* which only allowed for the user to play the machine. The computer programs developed and functions introduced in Chapter 16, "Reversi AI Simulation" allowed us to carry out our tests/experiments and to also see the results/analysis of these tests. Following Sweigerts suggestions we edited reversi.py to create a second version named *AISim1.py*. This version of the program enabled the computer to play itself. The *AISim1.py* program is the same as the original

Reversi program, except that the call to *getPlayerMove()* has been replaced with a call to *getComputerMove()*. This change can be seen in figures 3a and 3b.



```
if turn == 'X':
    # X's turn.
    otherTile = 'O'
    x, y = getPlayerMove(mainBoard, 'X')
    makeMove(mainBoard, 'X', x, y)
else:
    # O's turn.
    otherTile = 'X'
    x, y = getComputerMove(mainBoard, 'O')
    makeMove(mainBoard, 'O', x, y)
```

Figure 3a: Original Reversi program



```
if turn == 'X':
    # X's turn.
    otherTile = 'O'
    x, y = getComputerMove(mainBoard, 'X')
    makeMove(mainBoard, 'X', x, y)
else:
    # O's turn.
    otherTile = 'X'
    x, y = getComputerMove(mainBoard, 'O')
    makeMove(mainBoard, 'O', x, y)
```

Figure 3b: Changes made to reverse.py to create *AISim1.py*

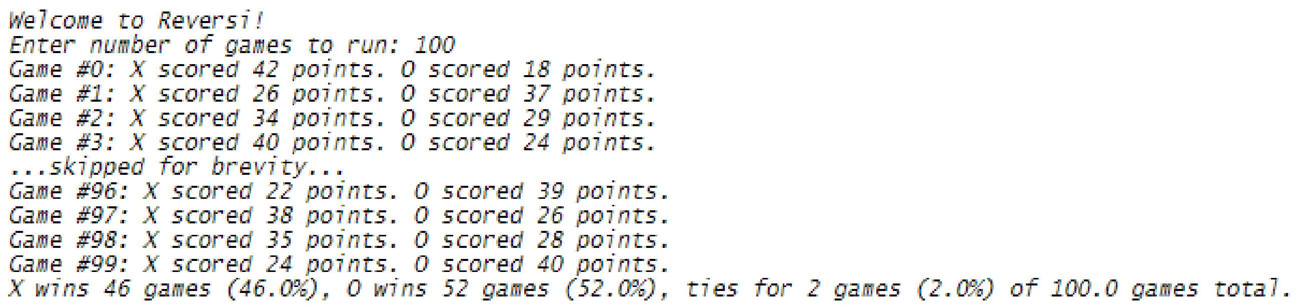
From here we edited *AISim1.py* to create *AISim2.py* which enabled the computer to play itself a number of times. The number of games played is determined by the user and produces results within 1-2 seconds per game such as the percentage and number of games won, lost or drawn by either X or O.

```
while True:
    # Reset the board and game.
    mainBoard = getNewBoard()
    resetBoard(mainBoard)
    if whoGoesFirst() == 'player':
        turn = 'X'
    else:
        turn = 'O'
    print('The ' + turn + ' will go first.')

    while True:
        drawBoard(mainBoard)
        scores = getScoreOfBoard(mainBoard)
        print('X has %s points. O has %s points' % (scores['X'], scores['O']))
        input('Press Enter to continue. ')

        if turn == 'X':
            # X's turn.
            otherTile = 'O'
            x, y = getComputerMove(mainBoard, 'X')
            makeMove(mainBoard, 'X', x, y)
        else:
            # O's turn.
            otherTile = 'X'
            x, y = getComputerMove(mainBoard, 'O')
            makeMove(mainBoard, 'O', x, y)

        if getValidMoves(mainBoard, otherTile) == []:
            break
        else:
            turn = otherTile

    # Display the final score.
    drawBoard(mainBoard)
    scores = getScoreOfBoard(mainBoard)
    print('X scored %s points. O scored %s points. ' % (scores['X'], scores['O']))

    if not playAgain():
        sys.exit()
```

Figure 4a: *AISim1.py* before changes to create *AISim2.py*

```
xwins = 0
owins = 0
ties = 0
numGames = int(input('Enter number of games to run: '))

for game in range(numGames):
    print('Game #%s:' % (game), end=' ')
    # Reset the board and game.
    mainBoard = getNewBoard()
    resetBoard(mainBoard)
    if whoGoesFirst() == 'player':
        turn = 'X'
    else:
        turn = 'O'
    while True:
        if turn == 'X':
            # X's turn.
            otherTile = 'O'
            x, y = getComputerMove(mainBoard, 'X')
            makeMove(mainBoard, 'X', x, y)
        else:
            # O's turn.
            otherTile = 'X'
            x, y = getComputerMove2(mainBoard, 'O')
            makeMove(mainBoard, 'O', x, y)

        if getValidMoves(mainBoard, otherTile) == []:
            break
        else:
            turn = otherTile

    # Display the final score.
    scores = getScoreOfBoard(mainBoard)
    print('X scored %s points. O scored %s points.' % (scores['X'], scores['O']))

    if scores['X'] > scores['O']:
        xwins += 1
    elif scores['X'] < scores['O']:
        owins += 1
    else:
        ties += 1

numGames = float(numGames)
xpercent = round(((xwins / numGames) * 100), 2)
opercent = round(((owins / numGames) * 100), 2)
tiepercent = round(((ties / numGames) * 100), 2)
print('X wins %s games (%s%%), O wins %s games (%s%%), ties for %s games (%s%%) of
%s games total.' % (xwins, xpercent, owins, opercent, ties, tiepercent, numGames))
```

Figure 4b: *AISim2.py* after changes to *AISim1.py*



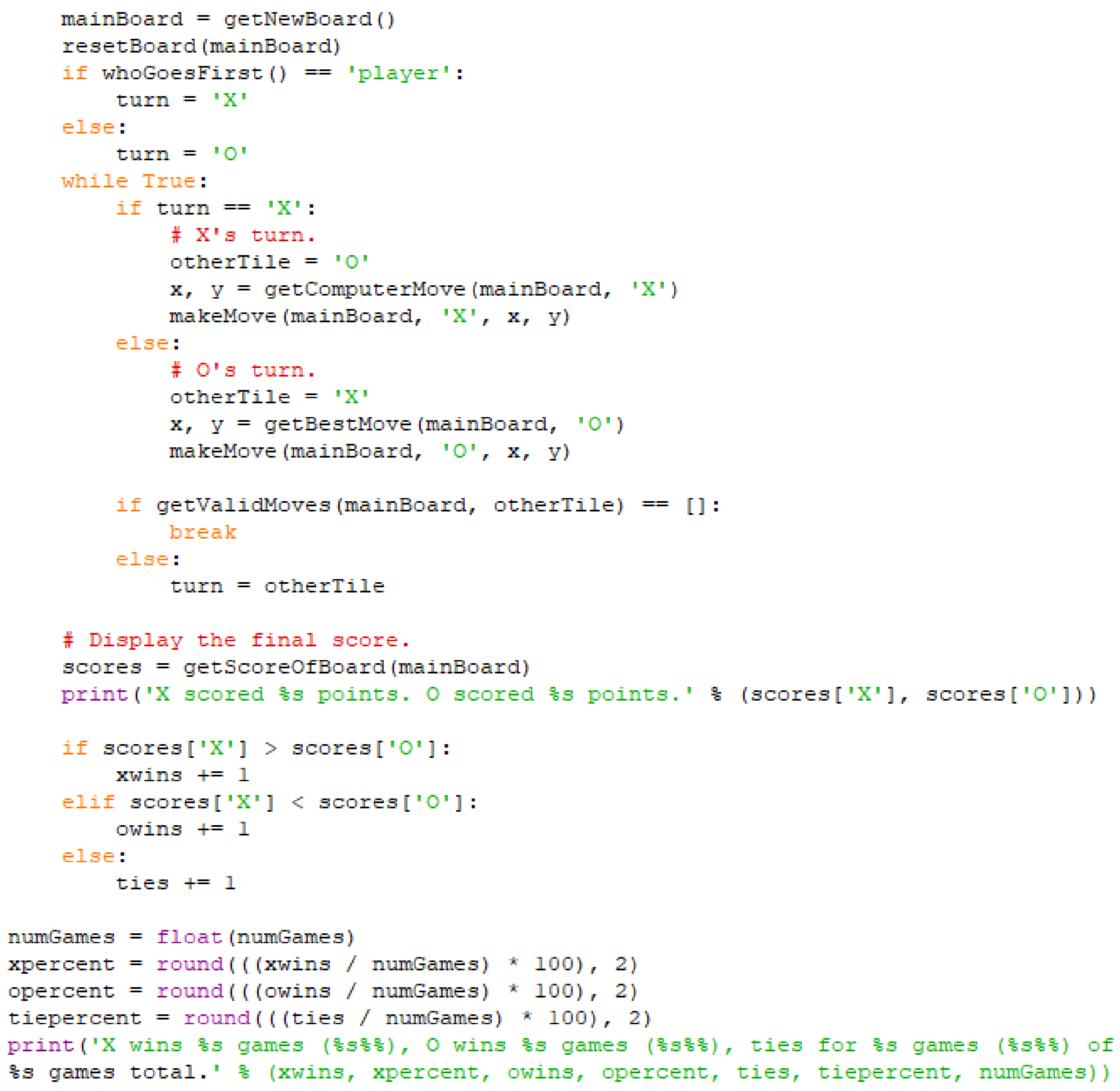
```
Welcome to Reversi!
Enter number of games to run: 100
Game #0: X scored 42 points. O scored 18 points.
Game #1: X scored 26 points. O scored 37 points.
Game #2: X scored 34 points. O scored 29 points.
Game #3: X scored 40 points. O scored 24 points.
...skipped for brevity...
Game #96: X scored 22 points. O scored 39 points.
Game #97: X scored 38 points. O scored 26 points.
Game #98: X scored 35 points. O scored 28 points.
Game #99: X scored 24 points. O scored 40 points.
X wins 46 games (46.0%), O wins 52 games (52.0%), ties for 2 games (2.0%) of 100.0 games total.
```

Figure 4c: Example of results obtained from *AISim2.py*

We finally edited *AISim2.py* to create *AISim3.py* which added some new functions with new algorithms. With these new additional functions, we will test, examine and analyse their performance and also use these functions to test our own heuristic and analyse the results and its performance compared to Sweigerts original heuristics. The new functions and algorithms can be seen in figure 5.

```
def getRandomMove(board, tile):
    # Return a random move.
    return random.choice( getValidMoves(board, tile) )

def isOnSide(x, y):
    return x == 0 or x == 7 or y == 0 or y ==7

def getCornerSideBestMove(board, tile):
    # Return a corner move, or a side move, or the best move.
    possibleMoves = getValidMoves(board, tile)

    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    # always go for a corner if available.
    for x, y in possibleMoves:
        if isOnCorner(x, y):
            return [x, y]

    # if there is no corner, return a side move.
    for x, y in possibleMoves:
        if isOnSide(x, y):
            return [x, y]

    return getComputerMove(board, tile)

def getSideBestMove(board, tile):
    # Return a side move, or then the best move.
    possibleMoves = getValidMoves(board, tile)
    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    # return a side move, if available
    for x, y in possibleMoves:
        if isOnSide(x, y):
            return [x, y]


    return getComputerMove(board, tile)

def getBestMove(board, tile):
    # Return best positions, or then the best move.
    possibleMoves = getValidMoves(board, tile)

    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    #return best positions including corner and two tiles away from corner.
    for x, y in possibleMoves:
        if bestPositions(x, y):
            return [x, y]

    return getComputerMove(board, tile)

def getWorstMove(board, tile):
    # Return the move that flips the least number of tiles.
    possibleMoves = getValidMoves(board, tile)
```

```python
    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    # Go through all the possible moves and remember the worst scoring move
    worstScore = 64
    for x, y in possibleMoves:
        dupeBoard = getBoardCopy(board)
        makeMove(dupeBoard, tile, x, y)
        score = getScoreOfBoard(dupeBoard)[tile]
        if score < worstScore:
            worstMove = [x, y]
            worstScore = score

    return worstMove

def getCornerWorstMove(board, tile):
    # Return a corner, or then the move that flips the least number of tiles.
    possibleMoves = getValidMoves(board, tile)

    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    # always go for a corner if available.
    for x, y in possibleMoves:
        if isOnCorner(x, y):
            return [x, y]

    return getWorstMove(board, tile)


print('Welcome to Reversi!')

xwins = 0
owins = 0
ties = 0
numGames = int(input('Enter number of games to run: '))

for game in range(numGames):
    print('Game #%s:' % (game), end=' ')
    # Reset the board and game.
```



```
    mainBoard = getNewBoard()
    resetBoard(mainBoard)
    if whoGoesFirst() == 'player':
        turn = 'X'
    else:
        turn = 'O'
    while True:
        if turn == 'X':
            # X's turn.
            otherTile = 'O'
            x, y = getComputerMove(mainBoard, 'X')
            makeMove(mainBoard, 'X', x, y)
        else:
            # O's turn.
            otherTile = 'X'
            x, y = getBestMove(mainBoard, 'O')
            makeMove(mainBoard, 'O', x, y)

        if getValidMoves(mainBoard, otherTile) == []:
            break
        else:
            turn = otherTile

    # Display the final score.
    scores = getScoreOfBoard(mainBoard)
    print('X scored %s points. O scored %s points.' % (scores['X'], scores['O']))

    if scores['X'] > scores['O']:
        xwins += 1
    elif scores['X'] < scores['O']:
        owins += 1
    else:
        ties += 1

numGames = float(numGames)
xpercent = round(((xwins / numGames) * 100), 2)
opercent = round(((owins / numGames) * 100), 2)
tiepercent = round(((ties / numGames) * 100), 2)
print('X wins %s games (%s%%), O wins %s games (%s%%), ties for %s games (%s%%) of
%s games total.' % (xwins, xpercent, owins, opercent, ties, tiepercent, numGames))
```

Figure 5: New functions and algorithms added to *AISim2.py* to create *AISim3.py*

4. Examination of existing heuristics elements

This section examines the existing heuristic strategies and move sets developed by Sweigert and tests them against *getComputerMove()* to see how well they perform. By analysing how each strategy is utilized by the system, this will provide us with adequate information and allow such heuristics to be individually ranked with respect to performance from most optimal to least optimal. We ran our tests over 200 games in total.

Test 1:

getComputerMove()* vs. *getComputerMove()

X – getComputerMove()

O – getComputerMove()

X wins 100 games (50.0%), O wins 91 games (45.5%), ties for 9 games (4.5%) of 200.0 games total.

After running tests of 200 games in total we observed that *getComputerMove()* vs. *getComputerMove()* returned similar results over the 200 games. The two heuristics are exactly the same and implement the same strategy of always going for a corner square if available and if no corner squares are available, *flank* the most letters of the opponent. Upon comparing *getComputerMove()* vs. *getComputerMove()*, the results obtained were as expected and one heuristic is no better than the other as they are identical.

Test 2:

getRandomMove()* vs. *getComputerMove()

X – *getComputerMove()*

O – *getRandomMove()*

X wins 181 games (90.5%), O wins 18 games (9.0%), ties for 1 games (0.5%) of 200.0 games total.

After comparing these heuristics, it is clear that *getRandomMove*() is not an optimal strategy as *getComputerMove()* wins by a substantial amount every time. The unpredictable nature of *getRandomMove*() has shown that it will lose nearly every time when pitted against a heuristic that has a strategic advantage.

Test 3:

getCornerSideBestMove()* vs. *getComputerMove()

X – *getComputerMove()*

O – *getCornerSideBestMove()*

X wins 138 games (69.0%), O wins 61 games (30.5%), ties for 1 games (0.5%) of 200.0 games

Unexpectedly, *getCornerSideBestMove()* is not the optimal strategy against *getComputerMove()* as the latter wins by a considerable margin after 200 simulated tests. We've determined that choosing side spaces over a space that *flanks* more letters is a bad strategy to use.

Test 4:

getWorstMove()* vs *getComputerMove()

X – *getComputerMove()*

O – *getWorstMove()*

X wins 199 games (99.5%), O wins 1 games (0.5%), ties for 0 games (0.0%) of 200.0 games total.

As clearly displayed above, the *getWorstMove()* algorithm loses almost 100% of the time against *getComputerMove()* because it makes a move that aims to flip over the least amount of tiles. It is surprising that this algorithm has won at all.

Test 5:

getCornerWorstMove()* vs *getComputerMove()

X – *getComputerMove()*

O – *getCornerWorstMove()*

X wins 184 games (92.0%), O wins 13 games (6.5%), ties for 3 games (1.5%) of 200.0 games total.

As displayed above, the *getCornerWorstMove*() is the same as *getWorstMove()* except it takes any available corner squares before taking the worst move. It still loses most of the games but seems to win a few more than the *getWorstMove()* algorithm. This shows that taking corner squares when they are available appears to be a better strategy.

Test 6:

getSideBestMove()* vs *getComputerMove()

X – *getComputerMove()*
O – *getSideBestMove()*

X wins 147 games (73.5%), O wins 49 games (24.5%), ties for 4 games (2.0%) of 200.0 games total.

getSideBestMove() takes a side space if there is one available, if not then it uses the same strategy as the *getComputerMove()* algorithm. Then this means side spaces are chosen before corner spaces. The *getSideBestMove()* algorithm still loses the majority of games against the *getComputerMove()*.

5. Examination of new heuristic element

From section 4, we have determined that *getComputerMove()* is the most optimal existing heuristic developed by Sweigert considering that the strategy beats all other heuristics. *getComputerMove()* always goes for a corner if available and if there are no corner moves available the strategy goes through all of the possible moves it can make and remembers the highest scoring move. Now, we are presented with the challenge of creating a new heuristic based on this analysis. The most optimal strategy in Reversi is to obtain the corners because once a corner or more than on corner is obtained, it cannot be *flanked* and this gives the player a positional advantage over his/her opponent.

Given this idea, we created our own heuristic named *getBestMove()*. *getBestMove()* takes the idea of always trying to obtain a corner and if no corner square is available, go through all of the possible moves it can make and remember the highest scoring move, but adds an extra element to this strategy. As well as trying to obtain the corner squares if they are available, our heuristic *getBestMove(),* also tries to obtain three crucial positions around the corner squares. These positions include (3,3), (3,1) and (1,3) positioned around the top left square (1,1). (8,3), (6,3) and (6,1) positioned around the top right square (8,1). (3,6), (3,8) and (1,6) positioned around the bottom left square (1,8) and finally (6,6), (8,6) and (6,8) positioned around the bottom right square (8,8). These crucial positions can be seen in figure 6a.

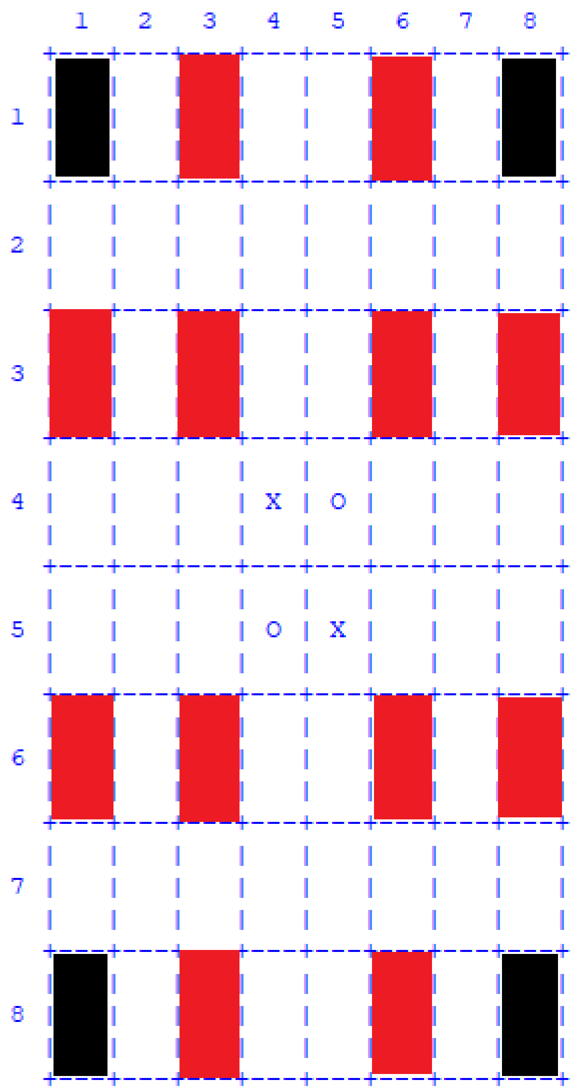


Figure 6a: Highlighted red squares show the best positions to take based around the corner squares

The idea behind this strategy is that if these positions are obtained it is more likely to lead to obtaining a corner square which we know is the most optimal strategy, and the more corners that are obtained the greater the positional advantage that player has. We will refer to these positions as “optimal spots”. The basic principle of this strategy is to avoid playing your letter or “piece” next to a corner, we will refer to these positions as “danger spots”, see figure 6b, and if possible, force the opponent to play their letter next to the corner or in the “danger spots”. The positions highlighted in red in figure 6 allow the player to control access to the corners. The corners are the most important positions to obtain and the squares controlling the corners are where you want to get early in the game to gain a positional advantage as if your opponent *flanks* these squares, he/she is now in a “danger spot” and you are more likely to take the corner square.

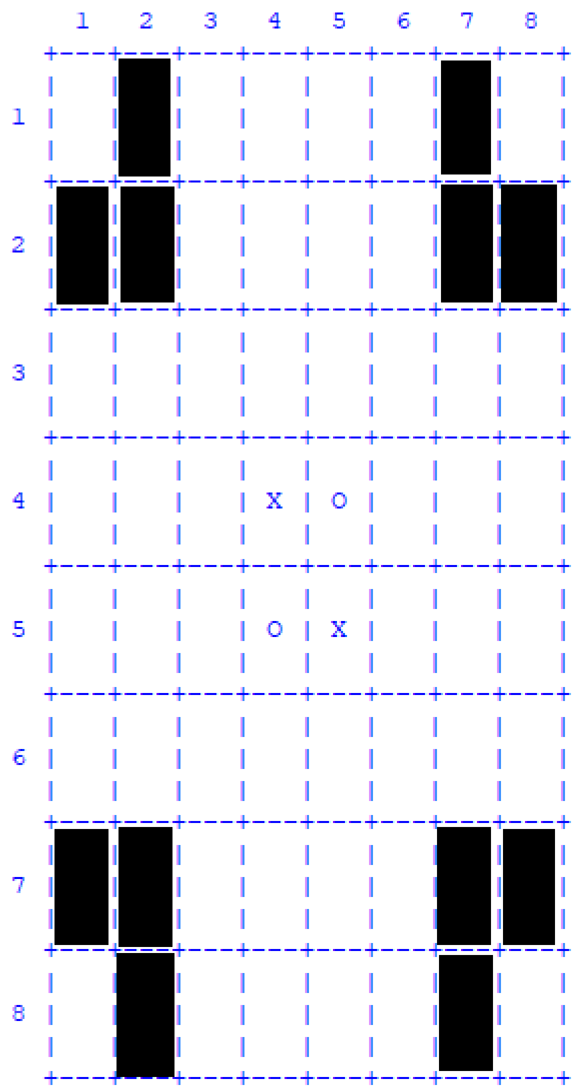
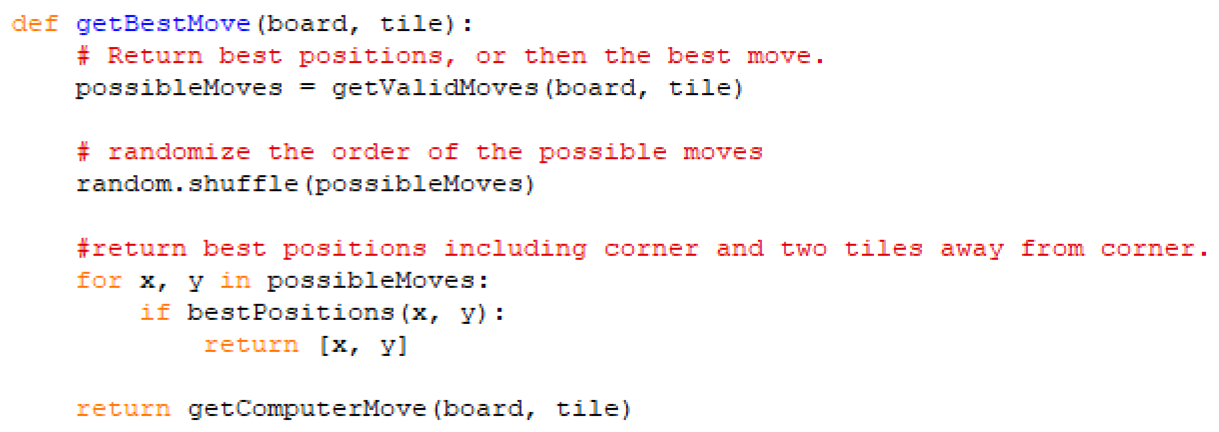


Figure 6b: Danger spots can be seen are highlighted in black

The code used to create our new heuristic *getBestMove()* can be seen in figure 7a and 7b.

```
def bestPositions(x, y):
    # Returns true if the positions are two tiles away from corner tiles.
    return (x == 2 and y == 2) or (x == 2 and y == 0) or (x == 0 and y == 2) or(x == 0 and y == 0) or
    (x == 5 and y == 2) or (x == 5 and y == 0) or (x == 7 and y == 2) or (x == 7 and y == 0) or
    (x == 2 and y == 5) or (x == 2 and y == 7) or (x == 0 and y == 5) or (x ==0 and y == 7) or
    (x == 5 and y == 5) or (x == 5 and y == 7) or (x == 7 and y == 5) or (x == 7 and y == 7)
```

Figure 7a: The function used by getBestMove() to obtain the most optimal positions



```python
def getBestMove(board, tile):
    # Return best positions, or then the best move.
    possibleMoves = getValidMoves(board, tile)

    # randomize the order of the possible moves
    random.shuffle(possibleMoves)

    #return best positions including corner and two tiles away from corner.
    for x, y in possibleMoves:
        if bestPositions(x, y):
            return [x, y]

    return getComputerMove(board, tile)
```

Figure 7b: Showing getBestMove() function

To determine the legitimacy of this heuristic we tested *getBestMove()* against Sweigerts original heuristics and analysed the results. We ran our test over 200 games in total.

5.1 Testing Our New Heuristic *getBestMove()* Against Sweigerts Original Heuristics

Test 1:

getSideBestMove()* vs. *getBestMove()

X – *getSideBestMove()*

O – *getBestMove()*

X wins 25 games (12.5%), O wins 171 games (85.5%), ties for 4 games (2.0%) of 200.0 games total.

Test 2:

getCornerSideBestMove()* vs. *getBestMove()

X – *getCornerSideBestMove()*

O – *getBestMove()*

X wins 34 games (17.0%), O wins 163 games (81.5%), ties for 3 games (1.5%) of 200.0 games total.

Test 3:

getRandomMove()* vs. *getBestMove()

X – *getRandomMove()*

O – *getBestMove()*

X wins 15 games (7.5%), O wins 180 games (90.0%), ties for 5 games (2.5%) of 200.0 games total.

Test 4:

getWorstMove()* vs. *getBestMove()

X – *getWorstMove()*

O – *getBestMove()*

X wins 0 games (0.0%), O wins 199 games (99.5%), ties for 1 games (0.5%) of 200.0 games total.

Test 5:

getCornerWorstMove()* vs. *getBestMove()

X – *getCornerWorstMove()*

O – *getBestMove()*

X wins 7 games (3.5%), O wins 193 games (96.5%), ties for 0 games (0.0%) of 200.0 games total.

Test 6:

getComputerMove()* vs. *getBestMove()

X – *getComputerMove()*

O – *getBestMove()*

X wins 86 games (43.0%), O wins 105 games (52.5%), ties for 9 games (4.5%) of 200.0 games total.

Test 7:

getBestMove()* vs. *getBestMove()

X – *getBestMove()*

O – *getBestMove()*

X wins 198 games (49.5%), O wins 187 games (46.75%), ties for 15 games (3.75%) of 400.0 games total.

5.2 Analysis of Test Results

Upon testing our heuristic *getBestMove*() against Sweigerts existing heruistics we have determined that *getBestMove()* is the most optimal strategy out of all of the strategies tested. *GetBestMove()* beats all of the existing heuristics developed by Sweigert. We expected our heuristic to beat most of Sweigerts heuristics as *getBestMove()* implements the most optimal strategy of always going for a corner square if available and in section 4, from our tests, we saw that this strategy was the most optimal one as once a corner square is obtained, it cannot be *flanked* and the player has also gained a positional advantage over his/her opponent. But the true test for our heuristic was pitting it against *getComputerMove().*

GetComputerMove() was originally the most optimal strategy developed by Sweigert that beat all of the other existing heuristics. The results from our first set of tests, in section 4, prove this statement. We were unsure of how well our new heuristic would perform against Sweigerts *getComputerMove()*, but upon testing getBestMove() vs. getComputerMove() we can see that getBestMove() does in fact beat getComputerMove(). Over 200 simulated games in total, getBestMove() won 105 games (52.5%) compared to getComputerMove() which only won 86 games (43.0%). They tied for 9 games in total (4.5%). Our tests of *getBestMove()* proved successful as it performed the best out of all of the existing heuristics developed by Sweigert.

6. Evaluation and Summary

This paper tried to evaluate the performance of various heuristics developed by Al Sweigert in his book “Invent with Python” (2008). The project was based on the 8x8 board game Reversi which allowed us to tractably explore heuristics as the complexity of the game is not too great nor too small. We examined, implemented and tested existing heuristics and analysed their performance. From our tests we found that always going for a corner square was the most optimal strategy as it provided the player with a greater positional advantage over his/her opponent. It was also interesting to note that although obtaining the corners was the most optimal heuristic, obtaining the three “optimal spots” around the corner also played a major role in improving this heuristic.