Matthew James D. Villarica – S11

MCO2: Casual Gamer

1. Representation of State

```
2. public class Node {
3.
4.     //deep copy of the configuaration of the board
5.     private Board board;
6.
7.     //possible moves player can make
8.     private ArrayList<Move> moves;
9.
10.    //the player on this state, represented as an enum that can either be COMPUTER or
   HUMAN
11.    private PlayerSide player;
12.
13.    //the value of a cut-off node, higher utility favors COMPUTER
14.    private double utility;
15.
16.    //move to get to this state
17.    private Move move;
```

Figure 1. Code snippet of the Node class, the representation of each state.

Each state is represented as a node data structure that has the pertinent information and attributes for each board configuration in a checkers game tree. The first attribute is the board, which contains an 8 x 8 array of squares that represent the board itself and the corresponding pieces and their respective locations in the 2D array that represents the board in that particular state. The board also contains a list of pieces for each player, that point to the same pieces that are present in the 2D array of squares that represent the board.

```
1. import java.util.*;
2. public class Board {
3.     //8 x8 board for gameplay
4.     private Square[][] board;
5.
6.     //list of pieces for the maximizer, COMPUTER
7.     private ArrayList<Piece> compPieces;
8.     //list of pieces for the minimizer, HUMAN
9.     private ArrayList<Piece> humanPieces;
```

Figure 2. Code snippet of the Board class, an attribute in each Node.

The board also contains a list of pieces for each player, that point to the same pieces that are present in the 2D array of squares that represent the board.

```java
public class Square {
    //if there is a piece, a reference to the Piece on the board
    private Piece piece;
    //row position on board
    private int row;
    //column position on board
    private int col;
```

```java
public class Piece {

    //allegiance of piece, either COMPUTER or HUMAN
    private PlayerSide side;
    //row position on board
    private int row;
    //col position on board
    private int col;
    //print this symbol for representing the piece on the board
    private String symbol;
    //false if this piece is not a king, true otherwise
    private boolean isKing;
```

Figure 3. Code snippets of Piece and Square attributes.

Both classes have a row and column attribute. PlayerSide is an enum that can be either HUMAN or COMPUTER, which is used to differentiate the minimizer and maximizer throughout the code base.

```java
public class Move {

    //directions this move contains, can contain UPRIGHT, UPLEFT, DOWNLEFT, DOWNRIGHT
    private ArrayList<Direction> directionMoves;

    //type of move represented as an enum that can be JUMP (includes MULTIPLE JUMP) or STANDARD
    private MoveType type;

    //reference to the piece being moved
    private Piece movePiece;

    //value if it is evaluated by the utility function
    private double value;

    //parent move to get to this move
    private Move parent;

    //true if this will lead to the piece transforming to a king, false otherwise
    private boolean transformToKing = false;
```

Figure 4. Code snippets of Move class.

The list called moves in the Node class contains a list of valid moves or actions that can be done by the player of that state. This is regenerated for every unique configuration of the board since every

state has a unique set of moves that can be done for that state. Each move has a parent move, which is how the initial move to make is retrieved after minimax search returns a move at the leaf node. A move's value is always the same as the value computed by the utility function of the state resulting from making that specific move.

The player represents who will be playing at each state. That is, the generated actions in that particular state are moves that can only be done by the player of that node. The utility is a double that receives the value computed by the evaluation function when the minimax algorithm decides to cut off search at the max depth, or when it encounters a terminal node. The last attribute called move is the legal move that was executed in order to arrive to that particular node.

```
1. if(turn == PlayerSide.COMPUTER) {
2.     Move comp;
3.
4.     Node root = new Node(board, true);
5.     comp = root.MinMaxSearchMAX();
6.
7.     while(comp.getParent() != null) {
8.
9.           comp = comp.getParent();
10.
11.    }
12.
13.    System.out.println("*AI's choice: " + comp);
14.
15.
16.    board.executeMove(comp);
```

Figure 5. Instantiation of game tree by calling search algorithm.

The representation of the tree is represented as instantiating the root node given the initial state, which is the current board configuration of the ongoing game, and new nodes are generated and the tree is explored through the minimax search.

2. Utility Function

```
1. /**
2. * This method is the heuristic evaluation function and the utility function for nodes
   at the
3. * cut off depth or nodes that are leaf nodes.
4. *
5. * @param newNode the node being evaluated
6. * @return a heuristic value of the state, a higher value favors COMPUTER, a smaller
   value favors HUMAN the minimizer
7.      */
8.     public double utility(Node newNode) {
9.         //total value computed by eval function
10.        double utility;
11.        //difference of piece score of COMPUTER and HUMAN feature
12.        double pieceDifference;
13.        //center control feature
```

```
14.         double numCenter;
15.         //back piece for king denial feature
16.         double numBackPieces;
17.         //vulnerable pieces feature
18.         double numVulnerable;
19.
20.
21.         /*First, check if node is a terminal state where either HUMAN or COMPUTER
    wins. */
22.         if(newNode.getBoardConfig().getHumPieces().size() == 0 || (player ==
    PlayerSide.HUMAN && newNode.getMoves().size() == 0))
23.             utility = 90;
24.         else if(newNode.getBoardConfig().getCompPieces().size() == 0 || (player ==
    PlayerSide.COMPUTER && moves.size() == 0))
25.             utility = -90;
26.         else {
```

Figure 6. Code snippet of first few lines of utility function.

The evaluation function takes a node as a parameter, which contains the state that will be evaluated, and returns a double after getting the sum of all the different features. The utility value is composed of 4 features: the difference in the piece score, the amount of pieces in the center, the key back pieces that defend against kings, and the amount of capturable pieces in this state which leads to a loss in either normal pieces or kings. The first part of the utility function checks if the current state is a node that leads to a win or a loss for each player. If either player runs out of moves, or has no pieces that have not been captured yet, then that state is a terminal node and the appropriate value is assigned to utility depending on whether it is the loss of the computer or human.

```
1. else {
2.
3.        /*Evaluate if it is not a terminal node with the heuristic */
4.         int compPieceScore = 0;
5.         int humanPieceScore = 0;
6.
7.        //compute piece score for computer with corresponding weights
8.         for(int i = 0; i < newNode.getBoardConfig().getCompPieces().size(); i++) {
9.             if(newNode.getBoardConfig().getCompPieces().get(i).isKing())
10.                compPieceScore += kingW;
11.            else compPieceScore += normalW;
12.
13.        }
14.
15.        //compute piece score for human with corresponding weights
16.        for(int i = 0; i < newNode.getBoardConfig().getHumPieces().size(); i++) {
17.            if(newNode.getBoardConfig().getHumPieces().get(i).isKing())
18.                humanPieceScore += kingW;
19.            else humanPieceScore += normalW;
20.
21.        }
22.        //more pieces/kings puts you at an advantageous position
23.        pieceDifference = compPieceScore - humanPieceScore;
```

Figure 7. Code snippet of piece score feature computation.

If the given node is not a state that leads to a terminal state, then search was cut off at a certain depth to prevent the search from causing an overflow error from the unreasonable amount of recursive calls made to reach a terminal state. It was found that depth 7 is ideal in terms of time where the minimax algorithm's completion is still within the range of milliseconds. A non-terminal state is treated as a leaf node, and it is evaluated based on the heuristic evaluation function which contains the aforementioned features. The first feature is based off of the amount of normal pieces and kings each player has. It is generally more favorable for a player if they have more pieces to work with than the opponent, especially when it comes to kings since they can move both forward and downward the board (Buffington, 2015).

For each player, a piece score is computed by adding a value for each piece to the total piece score. Kings have higher weights compared to normal pieces. Through trial and error it was found that the ideal weights are for a normal piece to be worth around 60% to 65% worth of a king. So the value assigned to king weights is 2, and 1.3 is the assigned weight for normal pieces. The difference of the two scores is calculated. A higher value for the difference in piece score favors COMPUTER because it means this particular state leads to captures of HUMAN pieces.

```
1. numCenter = 0;
2. numBackPieces = 0;
3.
4. if(newNode.getDestMove().getPiece().getSide() == PlayerSide.COMPUTER) {
5.
6.    //compute for center control feature
7.    for(int i = 3; i <= 4; i++) {
8.        for(int j = 2; j <= 5; j++) {
9.            if(newNode.getBoardConfig().getBoard()[i][j].getPiece() != null) {
10.
11.               if(newNode.getBoardConfig().getBoard()[i][j].getPiece().getSide() ==
   PlayerSide.COMPUTER)
12.                   numCenter += normalW / 2;
13.
14.           }
15.       }
16.    }


-------------------------------------------------------------------------------------

1. } else {
2.     //compute for center control feature
3.         for(int i = 3; i <= 4; i++) {
4.             for(int j = 2; j <= 5; j++) {
5.                 if(newNode.getBoardConfig().getBoard()[i][j].getPiece() != null) {
6.
7.                     if(newNode.getBoardConfig().getBoard()[i][j].getPiece().getSide()
   == PlayerSide.HUMAN)
8.                                 numCenter += normalW / 2;
9.
10.                }
11.            }
```

```
        }
//since HUMAN is minimizer, negative
numCenter *= -1;
```

Figure 8. Code snippet of center control feature computation.

The next feature is based off of the rule of “controlling the center makes you control the game”. According to Buffington (2015), controlling the 8 squares in the middle and putting your pieces there allows you to control the game more and is more likely to lead to favorable trades and captures if you have more pieces there. For every piece that is in those 8 squares, half the weight of a normal piece is added to the score for that player. Only half of the weight of the normal piece is used because this will avoid the player from unnecessarily putting vulnerable pieces to the center, and more weight should still be given to states that lead to captures because Checkers is a game that rewards calculated aggression.

```
//compute for back pieces feature
if(newNode.getBoardConfig().getBoard()[0][1].getPiece() != null)
    if(newNode.getBoardConfig().getBoard()[0][1].getPiece().getSide() ==
PlayerSide.COMPUTER)
            numBackPieces += normalW - 0.2;
    if(newNode.getBoardConfig().getBoard()[0][5].getPiece() != null)
       if(newNode.getBoardConfig().getBoard()[0][5].getPiece().getSide() ==
PlayerSide.COMPUTER)
            numBackPieces += normalW - 0.2;


-------------------------------------------------------------------------

//compute for back pieces feature
    if(newNode.getBoardConfig().getBoard()[7][2].getPiece() != null)
        if(newNode.getBoardConfig().getBoard()[7][2].getPiece().getSide() ==
PlayerSide.HUMAN)
            numBackPieces += normalW - 0.2;
    if(newNode.getBoardConfig().getBoard()[7][6].getPiece() != null)
        if(newNode.getBoardConfig().getBoard()[7][6].getPiece().getSide() ==
PlayerSide.HUMAN)
            numBackPieces += normalW - 0.2;
//invert sign, since human is MINIMIZER
  numBackPieces *= -1;
      }
```

Figure 9. Code snippet of computation of back pieces feature.

To deny the opponent from reaching the king row, a good strategy to use is to keep the two important pieces in the back row that can defend and deny any attempts at crowning a king (Allen, 2021). The location of these pieces on the board can be seen in the source code, for both the minimizer and maximizer respectively. These pieces at the back are assigned a weight slightly lower than the weight

of capturing standard piece. States with these defending pieces staying at the king row are desirable, but not as much as the weighting of capturing the opponents pieces and attempting to crown pieces yourself.

```
1. //count vulnerable pieces in this state, based off of amount of generated jump movesd
2.              double j;
3.              numVulnerable = 0;
4.              for(int i = 0; i < newNode.getMoves().size(); i++) {
5.
6.                  if(newNode.getMoves().get(i).getType() == MoveType.JUMP) {
7.                      j = (0.60 * normalW) *
   newNode.getMoves().get(i).getDirections().size();
8.                      numVulnerable -= j;
9.                  }
10.             }
11.
12.             //since HUMAN is MINIMIZER, invert sign
13.             if(newNode.getDestMove().getPiece().getSide() == PlayerSide.HUMAN)
14.                 numVulnerable *= -1;
15.
16.             utility = pieceDifference + numCenter + numBackPieces + numVulnerable;
17.
18.         }
19.         return utility;
20.
21.     }
```

Figure 10. Code snippet of computation of vulnerable pieces score.

The final feature that is calculated before the sum of all the features is computed is the amount of vulnerable pieces in this state. That is, the amount of pieces that can be captured through jump moves by the opponent in the next turn. This encourages some degree of conservative play, but rewards trading of pieces as long the resulting state is favorable to the player of this node. The weighting of conserving pieces is lower than the standard piece weight because aggression should be encouraged more, but a king should not be sacrificed unnecessarily if it can be avoided. Through testing, although it is not perfect, it was found that in most cases the agent would only put it's pieces in a vulnerable state if a trade was possible through forced jumps. Trading pieces and making necessary sacrifices is a great strategy that increases one's chances of winning (UltraBoardGames, n.d.).

3. Effect of Move Ordering

In order to implement the agent with move ordering, a simple move ordering function was made to sort the moves before the children of each node is explored. The move ordering function sorted the moves in such a way that the moves that lead to king transformations and the moves that lead to more pieces being captured were explored first. Its more likely that states that are a result of these moves lead to having the best states explored first, which can lead to more pruning.

Experiment 1: Agent VS Human

In the first experiment, the exact same sequence of moves are done against both the agent with move ordering and the agent without move ordering. The amount of nodes traversed was recorded through a static global variable that incremented for each recursive call of the minimax algorithm, and the execution time of the minimax algorithm was recorded as well.

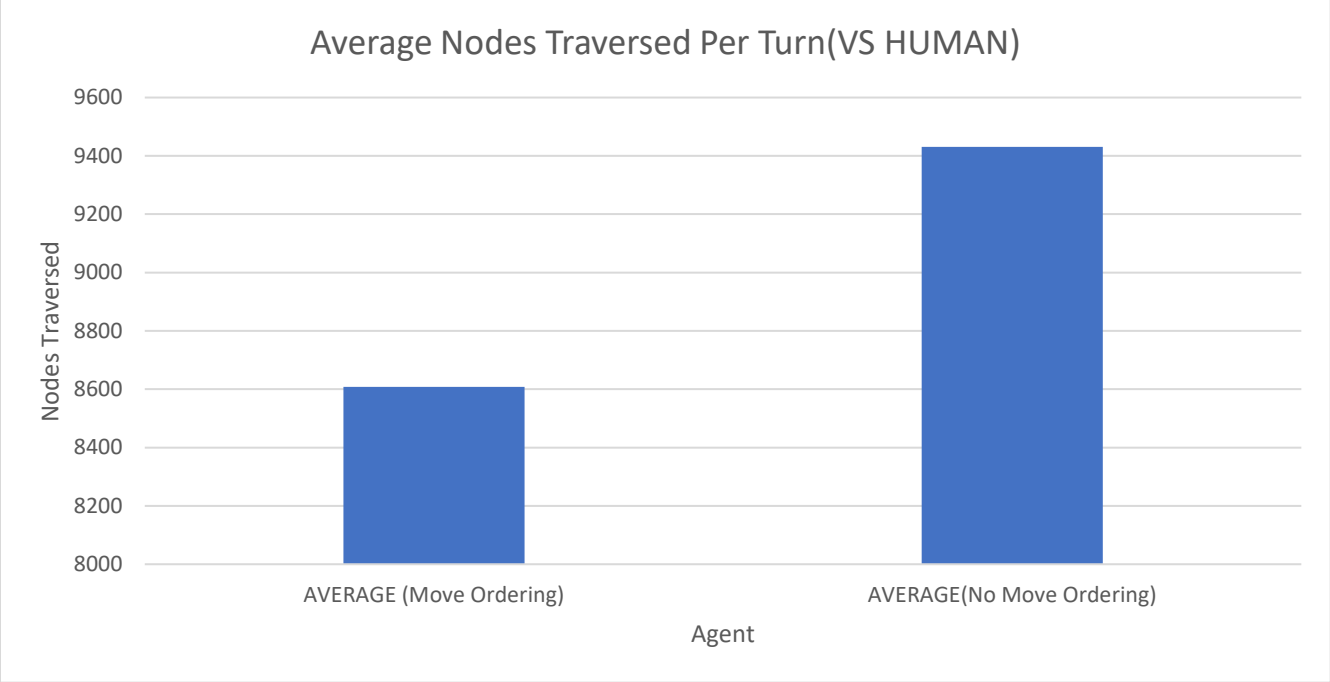

Figure 11. Average amount of nodes traversed for each search for both agents.

On average, 9430 nodes are traversed per search by the agent without move ordering and 8609 nodes are traversed on average by the agent with move ordering. Although it is not a significant amount, it can be seen that the agent with move ordering is more efficient when it comes to the amount of nodes it explores to arrive at the exact same solution.

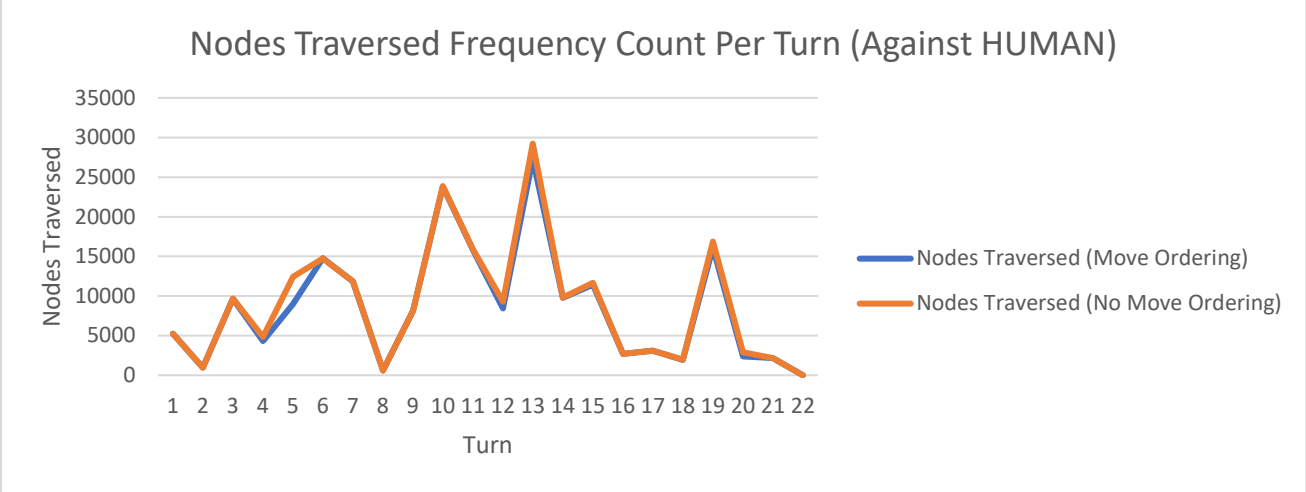

Figure 12. Amount of nodes traversed for each search in both agents.

Since the exact same moves are being done against both agents, the agent with move ordering and the agent without move ordering are essentially traversing the same tree. For every search, move ordering is only slightly more efficient compared to the agent without move ordering.

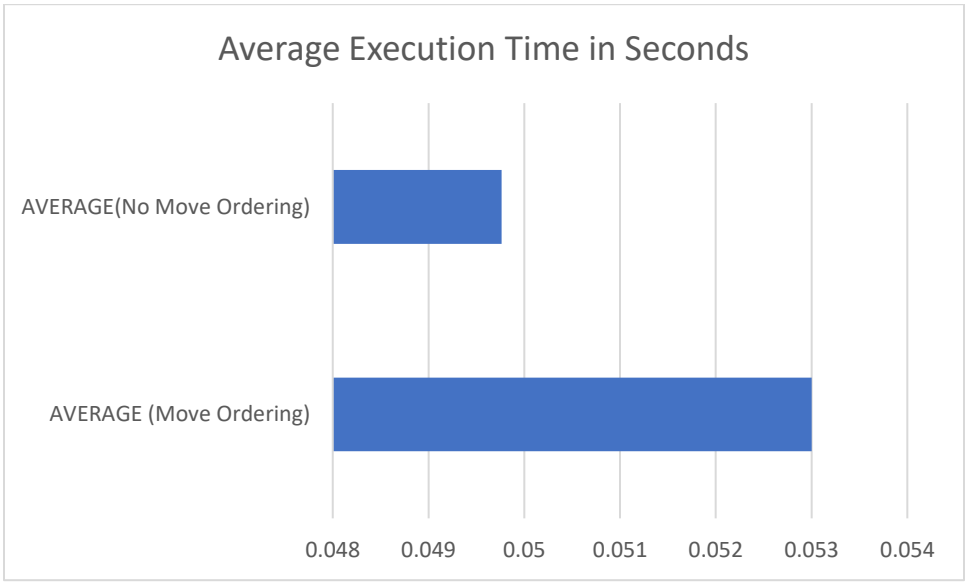

Figure 13. Average execution time in seconds for both agents.

It is expected that since move ordering lessens the amount of nodes traversed by the algorithm, execution time should in turn become smaller. But it was found that the call to the move ordering function for every recursive call only made the execution time slightly longer in the vast majority of cases unless a very significant amount of nodes have been pruned as a result of the move ordering. This may be because of the implementation of the function to sort the list of moves. The cost of time in

calling the function and the operations of sorting the move list outweighs the time reduced from alpha beta pruning.

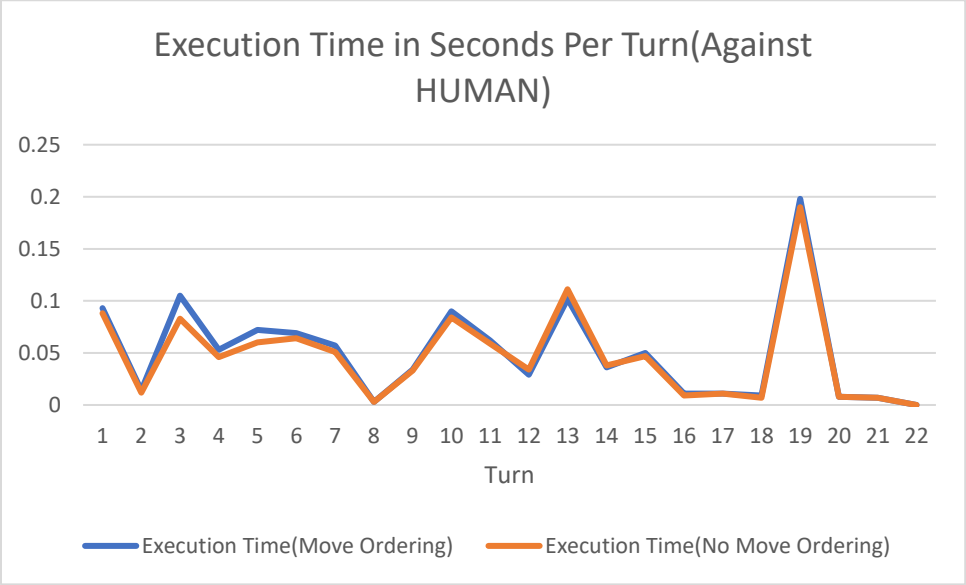

Figure 14. Execution time for each search.

Although execution time is slightly slower due to the move ordering function, the agent that implements move ordering follows vastly the same trend as the agent that does not have move ordering.

	Playing Against Human (Move Ordering)		Playing Against Human (No Move Ordering)	
Turn	Nodes Traversed	Execution Time (in seconds)	Nodes Traversed	Execution Time(in seconds)
1	5250	0.093	5244	0.088
2	959	0.014	956	0.012
3	9643	0.105	9698	0.083
4	4302	0.053	4856	0.046
5	9011	0.072	12438	0.06
6	14762	0.069	14740	0.064
7	11834	0.057	11880	0.051
8	583	0.003	584	0.003
9	8164	0.034	8159	0.033
10	23801	0.09	23856	0.084
11	15810	0.062	15916	0.059
12	8425	0.029	9287	0.034
13	27296	0.102	29233	0.111

14	9751	0.036	9817	0.038
15	11410	0.05	11669	0.047
16	2712	0.011	2712	0.009
17	3089	0.011	3098	0.011
18	1911	0.009	1960	0.007
19	16099	0.198	16885	0.19
20	2389	0.008	2880	0.008
21	2174	0.007	2176	0.007
22	2	0	2	0
AVERAGE	8608.045455	0.053	9430.666667	0.049761905

Table 1. Execution Time and Amount of Nodes Traversed Per Turn

Experiment 2: Agent (Move Ordering) VS Agent (No Move Ordering)

Since the sequence of moves done on both agents in the previous experiment is not a sequence of moves that demonstrates optimal play, the same experiment is done but the agent with move ordering has played against the agent without move ordering. Playing against another agent that plays optimally may be a more accurate representation of what the game tree generated would look like, which can generate different results.The agent with no move ordering is set as the maximizer, while the agent with move ordering is set as the minimizer. The minimizer goes first. Both agents take turns against each other until the agents start repeating the same move over and over again. Since the exact same heuristic evaluation function is used on both agents, the game gets to a point where both agents move the same pieces repeatedly in an infinite loop.

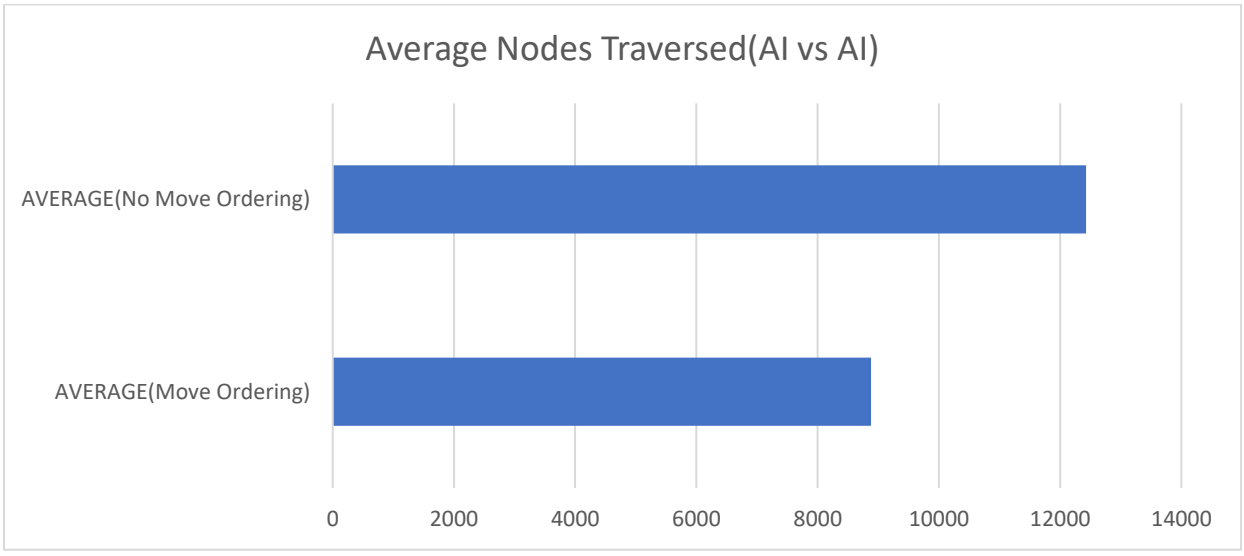

Figure 15. Average nodes traversed for agent against agent experiment.

In this experiment, it was also found that less nodes were traversed as a result of move ordering, but the difference is much larger compared to the previous experiment. On average, 8880 nodes were traversed per search for the agent with move ordering while the agent without move ordering traversed an average of 12426 nodes per search.

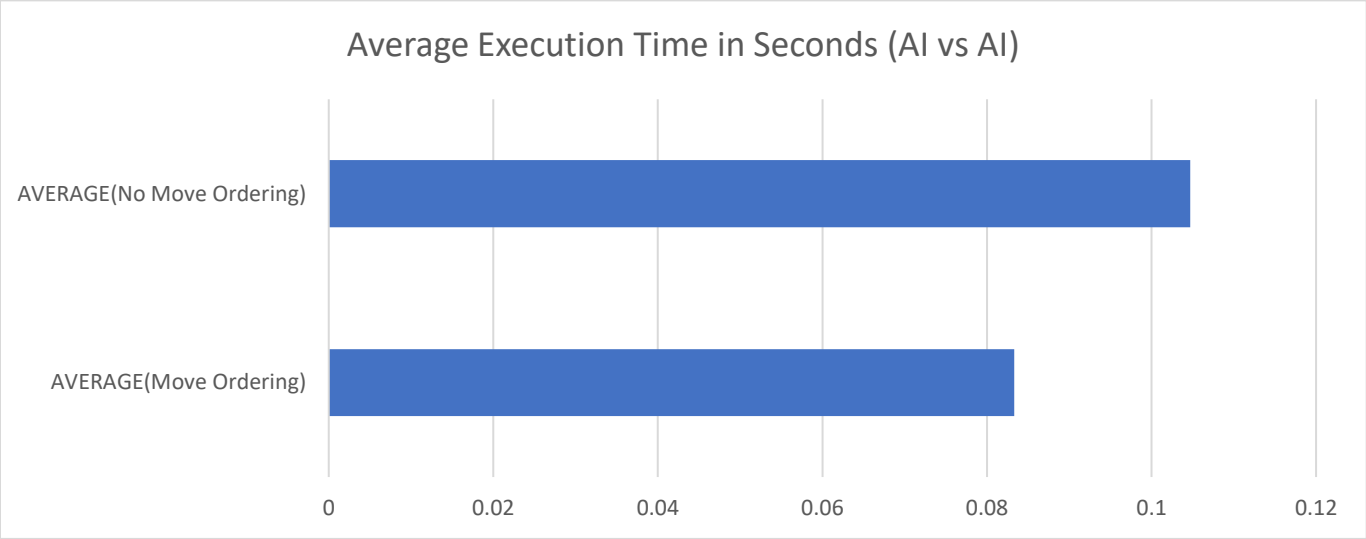

Figure 16. Average execution time in agent against agent experiment.

Unlike the previous experiment, the amount of nodes pruned now outweighs the computing time it takes to sort the moves before each move is explored. Although the difference is minimal, the agent with move ordering is faster compared to the agent without it when the average is considered.

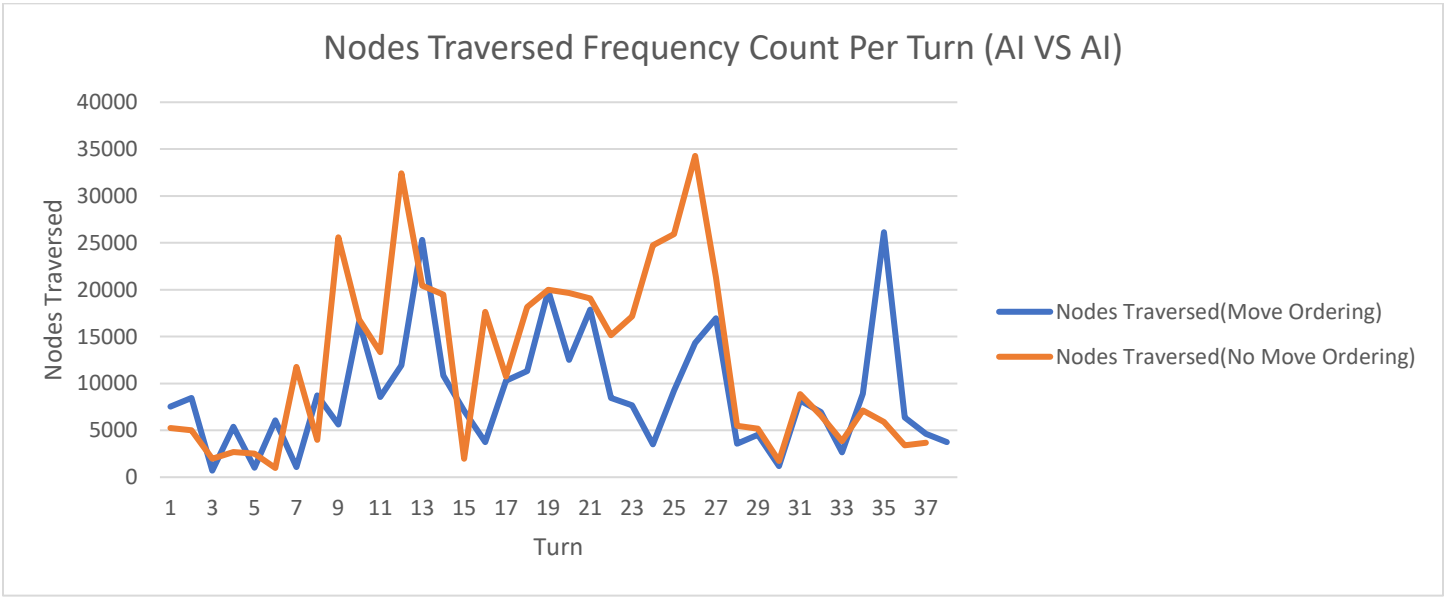

Figure 17. Nodes traversed per search in agent against agent experiment.

It is notable that the amount of nodes being pruned as a result of move ordering is consistently large from turn 15 to 27, which is mid-game to end-game. At this period of the game, the move ordering

function is more effective because this is where king transformations and multiple jump moves happen more often compared to early game. Since the best states are found earlier, more pruning is done during mid to end game in particular.

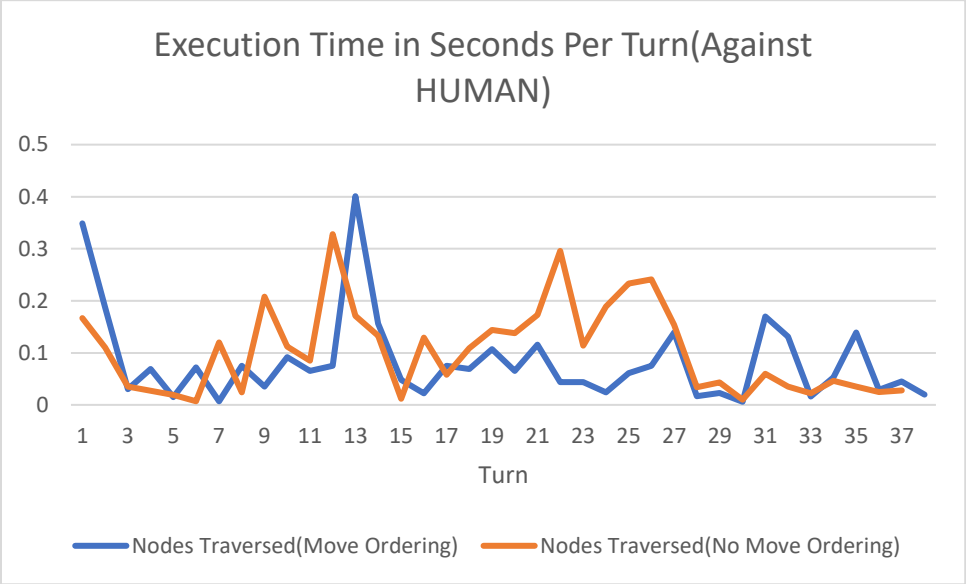

Figure 18. Execution time per search in agent against agent experiment.

The trend in the execution time is similar to the trend in the graph for nodes traversed for both agents. Turns 15 to 27 show a faster execution time because of the consistent pruning done during these turns.

	AI VS AI (Move Ordering)		AI VS AI(No Move Ordering)	
Turn	Nodes Traversed	Execution Time (in seconds)	Nodes Traversed	Execution Time(in seconds)
1	7536	0.349	5246	0.167
2	8439	0.187	5022	0.11
3	703	0.03	1969	0.035
4	5371	0.069	2701	0.027
5	1015	0.015	2503	0.019
6	6063	0.072	984	0.007
7	1072	0.007	11763	0.12
8	8712	0.075	3967	0.024
9	5611	0.035	25607	0.208
10	16566	0.092	16811	0.112
11	8545	0.065	13333	0.085
12	11936	0.075	32412	0.328
13	25302	0.401	20442	0.171
14	10838	0.156	19469	0.133

15	7090	0.048	1976	0.012
16	3732	0.022	17627	0.129
17	10302	0.075	10753	0.058
18	11339	0.069	18140	0.109
19	19883	0.107	19979	0.144
20	12516	0.065	19668	0.138
21	17861	0.116	19083	0.173
22	8465	0.044	15153	0.296
23	7655	0.044	17156	0.114
24	3508	0.024	24747	0.189
25	9253	0.061	25919	0.233
26	14318	0.075	34278	0.241
27	16939	0.139	21315	0.154
29	3582	0.017	5474	0.034
30	4542	0.023	5186	0.043
31	1166	0.006	1720	0.01
32	8188	0.17	8862	0.06
33	6963	0.131	6586	0.035
34	2661	0.016	3848	0.022
35	8908	0.053	7124	0.046
36	26134	0.139	5885	0.035
37	6355	0.029	3419	0.025
38	4628	0.045	3668	0.028
39	3753	0.02	N/A	N/A
AVERAGE	8880.263158	0.083315789	12426.89189	0.104702703

Table 2. Execution Time and Amount of Nodes Traversed Per Turn In Agent Against Agent Experiment

4. How close is the implementation of move ordering to perfect move ordering?

From both experiments, the simple move ordering function implemented has caused a generous amount of pruning to occur in the alpha beta search algorithm. However, when the averages of the nodes traversed are compared between the agent with move ordering and the agent without move ordering, the values are still very close to each other and the difference is minimal in the grand scheme of things, especially when only the first experiment is considered. In the first experiment, the agent with move ordering only explored 821 less nodes on average compared to the agent without move ordering. The implementation of move ordering is still very far when compared to the perfect case of move ordering of $O(b^{m/2})$, where a much more significant of nodes should be pruned. Some nodes are pruned, but the amount of pruning done is not enough to say that the move ordering function has made the algorithm significantly better than random move ordering with a complexity of $O(b^{3m/4})$.

To achieve the ideal branching factor of perfect move ordering, more advanced techniques of move ordering can be implemented in addition to the existing move ordering function, such as iterative deepening search and using transposition tables to cut off search at nodes already previously explored. Another sorting function may also be used during the early game, because the sorting of moves only prunes a significant amount of nodes during the mid to end game since this is when king transformations and multiple jumps are much more likely to happen, while king transformations are unlikely to happen in the early game.

Google Drive to Source Code
https://drive.google.com/drive/folders/1mwIbdHFNew4SGy_4RC_UltZyfNCF7AOo?usp=sharing

1	Skills I already possess which I used for the project	• Java • basic git version control
2	Skills I had to learn for the project	• How to use System.currentTimeMillis and DecimalFormat from Java API to record the execution time of the algorithm in seconds • How to use enums in Java properly
3	Challenges I encountered and how I solved it	• I encountered a problem with making deep copies of the board that didn't affect the original copy while generating and executing the moves in the minimax algorithm. In order to solve this, I had to look up how memory for reference types work exactly in Java. I learned that I mistakenly made everything point to the same object in memory because I thought I made a proper copy with the assignment "=" statement, so I had to instantiate new objects instead.

References

Buffington, E. (2015, March 6). *How to win in Checkers - YouTube*. https://www.youtube.com/watch?v=Lfo3yfrbUs0

Allen, H. (2021, October 22). *Checkers Strategy and Tactics: How to Win Every Time*. Hobby Lark. https://hobbylark.com/board-games/Checkers-Strategy-Tactics-How-To-Win