MCO2 : CASUAL GAMER

A major course output
for the course on
Introduction to Intelligent Systems
(CSINTSY)

Submitted by:

Capunitan, Jonaviene De Guzman

Submitted to:

Dr. Merlin Teodosia Suarez
Teacher

Jan 31, 2022

I. Introduction

The goal of the project was to explore and understand adversarial search problems by implementing a game-playing agent. To do this, we were to develop a Player vs. A.I. checkers game. The checkers agent must be able to make optimal decisions using the minimax algorithm with alpha-beta pruning.

II. The Checkers Game

Checkers is a two-player board game on an 8x8 checkerboard. Each player starts with 12 pawn pieces. The player with the black pieces goes first. Each player will take turns moving their pieces diagonally. A player wins when they capture all of their opponent's checkers, or when the opponent can no longer make any moves.

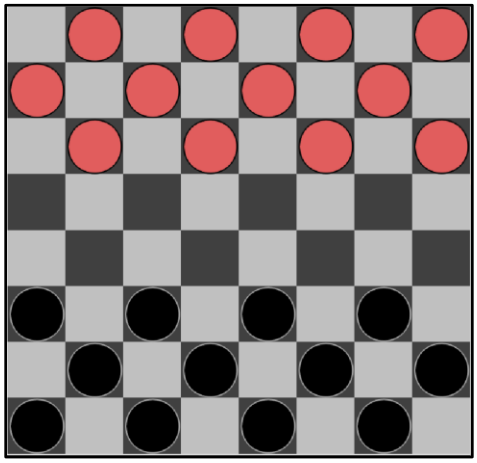


Figure 1. The initial state of the board

For moving and capturing, the game follows the standard British-American rules for checkers:

1. A pawn can move forward one tile diagonally if the chosen tile is not occupied.

2. Once a pawn reaches the row of the board, it becomes a king. A king moves the same way as a pawn, except it can move forward and backward.

3. If a player's piece is next to an opponent's piece, and the black tile behind the opponent's piece is open, the player must jump over it and remove the piece. A pawn can jump forward, while a king can jump forward and backward.

4. If it is possible for the player to make a jump, they have no choice but to take it. If one jump leads to another jump, the player must take that jump as part of the same turn. If the player reached the last row with a pawn, the player can no longer take

any further jumps. If there is more than one jump move, the player can choose which to take.

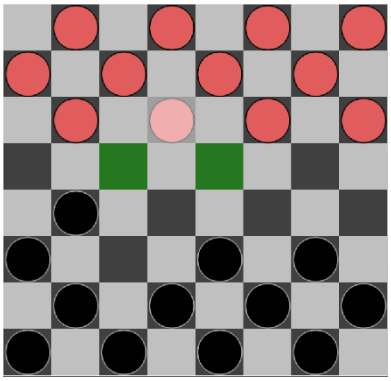


Figure 2.1. A possible move for pawn

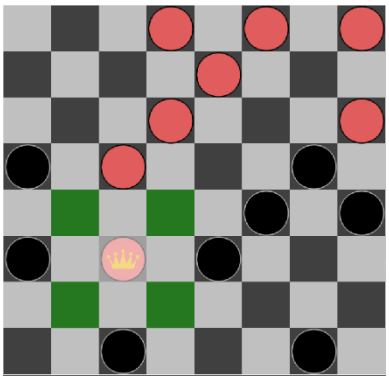


Figure 2.2. A possible move for king

For this game, the A.I. will always go first. Therefore, the A.I. player controls the black pieces and the human player controls the red pieces.

III. Implementation of the Game-Playing Agent

The game is developed in Java to utilize the language's OOP concepts and support for creating GUIs.

State Representation

Each state is represented by the Board class. The Board class contains the board itself as well as the positions of all the active pieces. Specifically, it contains an 8x8 array of Tile objects. Each Tile object represents a position on the board, and determines whether or not that position is occupied. Each Piece object contains its current position, its color, its type (pawn or king), and the board it is associated with. With this information, the Board class can generate all the possible moves for the current player, check for a game-over state, and measure the utility value of the board state.

```
public class Board {
    protected final int ROW = 8;
    protected final int COL = 8;
    protected Tile[][] board;

    private ArrayList<Piece> blackPieces;
    private ArrayList<Piece> redPieces;

  ...
```

Figure 3.1. The attributes/information of a board state

```
public class Tile {
    private final int ROW;
    private final int COL;
    private Piece occupant;

  ...
```

Figure 3.2. The attributes/information of a Tile

```
public class Piece {
    protected int row;
    protected int col;
    private boolean isKing;
    protected final boolean IS_BLACK;
    protected String file_path;
    public Board board;
  ...
```

Figure 3.3. The attributes/information of a Piece

To aid in generating a game tree, the Board class contains the following methods:

- *Board()* creates the initial state of the board. It calls the method initCheckers() to initialize the starting positions of each piece.

```
...
  public Board() {
        board = new Tile[ROW][COL];

        for (int i = 0; i < ROW; i++) {
            for (int j = 0; j < COL; j++)
                board[i][j] = new Tile(i, j, null);
        }
        blackPieces = new ArrayList<>();
        redPieces = new ArrayList<>();
        initCheckers();
    }
 ...
```

Figure 3.4. Board() creates the initial state of the board

- *Board(Board b)* duplicates the Board *b.* This is so that any simulated moves done by the minimax algorithm will not change the actual board state.

```
...
  public Board(Board b) {
        board = new Tile[ROW][COL];
        redPieces = new ArrayList<Piece>();
        blackPieces = new ArrayList<Piece>();

        for (int i = 0; i < ROW; i++) {
            for (int j = 0; j < COL; j++)
                board[i][j] = new Tile(b.board[i][j]);
        }

        for (int i = 0; i < b.redPieces.size(); i++) {
            Piece newPiece = new Piece(this, b.redPieces.get(i));
            redPieces.add(newPiece);
            board[newPiece.getRow()][newPiece.getCol()].setPiece(newPiece);
        }

        for (int i = 0; i < b.blackPieces.size(); i++) {
            Piece newPiece = new Piece(this, b.blackPieces.get(i));
            blackPieces.add(newPiece);
            board[newPiece.getRow()][newPiece.getCol()].setPiece(newPiece);
        }
    }
 ...
```

Figure 3.5. Board(Board b) duplicates a board state

- *getPointValue()* computes and returns the utility value of the board state. This will be discussed in a later section.

- *getLegalMoves(Board state, Boolean currentTurn)* returns all the possible moves for the current player. A checkers move contains the coordinates of initial position, the new position, and the positions of all captured pieces.

The method will first check if the current player has any forced jump moves by calling *getLegalJumpMoves().* This is done by looking for all of the current player's active pieces and checking if it is adjacent to any of the opponent's pieces. Once all the moves are generated, it will store the positions of each move's captured pieces.

```
...
    private ArrayList<CheckersMove> getLegalJumpMoves(Board state, boolean currentTurn) {
        Tile[][] stateBoard = state.board;
        ArrayList<CheckersMove> legalJumpMoves = new ArrayList<CheckersMove>();

        for (int i = 0; i < ROW; i++) {
            for (int j = 0; j < COL; j++) {
                if (stateBoard[i][j].isOccupied() && currentTurn == stateBoard[i][j].getPiece().isBlack()) {
                    if (stateBoard[i][j].getPiece().canJump(i - 2, j - 2, i - 1, j - 1, currentTurn)) {
                        legalJumpMoves.add(new CheckersMove(i, j, i - 2, j - 2, true));
                    }
                    if (stateBoard[i][j].getPiece().canJump(i - 2, j + 2, i - 1, j + 1, currentTurn)) {
                        legalJumpMoves.add(new CheckersMove(i, j, i - 2, j + 2, true));
                    }
                    if (stateBoard[i][j].getPiece().canJump(i + 2, j - 2, i + 1, j - 1, currentTurn)) {
                        legalJumpMoves.add(new CheckersMove(i, j, i + 2, j - 2, true));
                    }
                    if (stateBoard[i][j].getPiece().canJump(i + 2, j + 2, i + 1, j + 1, currentTurn)) {
                        legalJumpMoves.add(new CheckersMove(i, j, i + 2, j + 2, true));
                    }
                }
            }
        }

        // get the coords of each captured piece
        for (int i = 0; i < legalJumpMoves.size(); i++) {
            CheckersMove currMove = legalJumpMoves.get(i);
            int jumpRow = (currMove.getOldRow() + currMove.getNewRow()) / 2;
            int jumpCol = (currMove.getOldCol() + currMove.getNewCol()) / 2;
            currMove.addToCapture(stateBoard[jumpRow][jumpCol].getPiece());
        }

        return legalJumpMoves;
    }
...
```

Figure 3.6. getLegalJumpMoves(Board state, boolean currentTurn) generates the jump moves of the current player

After the jump moves are generated, it will call *updateJumpMoves()* to update each jump move to take any consecutive jumps. *updateJumpMoves()* is a recursive method that simulates each jump move, then checks if the moved piece can jump again by getting the piece's new jump moves. If the moved piece can still jump, it will call itself to update the move again. Otherwise, it will store that as the final jump move.

```
...
private ArrayList<CheckersMove> updateJumpMoves(Board state, boolean currentTurn, ArrayList<CheckersMove>
jumpMoves) {
        ArrayList<CheckersMove> maxJumpMoves = new ArrayList<CheckersMove>();
        for (int i = 0; i < jumpMoves.size(); i++) {
            Board tempBoard = new Board(state);
            int initRow = jumpMoves.get(i).getOldRow();
            int initCol = jumpMoves.get(i).getOldCol();
            Piece tempPiece = tempBoard.getPiece(initRow, initCol);
            boolean turnedKing = tempPiece.moveTo(jumpMoves.get(i)); // checks if the piece is newly crowned
            ArrayList<int[]> initCapturedPieces = jumpMoves.get(i).getCapturedPieces();
            ArrayList<CheckersMove> tempMoves =
              tempPiece.getLegalPieceMoves(getLegalJumpMoves(tempBoard, currentTurn));
            // if the piece can still jump, call this method again
            // otherwise, add current move as the max jump move
            if (tempMoves.size() != 0 && !turnedKing) {
                for (int j = 0; j < tempMoves.size(); j++)
                    tempMoves.get(j).addToCapture(initCapturedPieces);

                tempMoves = updateJumpMoves(tempBoard, currentTurn, tempMoves);
                for (int j = 0; j < tempMoves.size(); j++) {
                    tempMoves.get(j).changeInit(initRow, initCol);
                    maxJumpMoves.add(tempMoves.get(j));
                }
            } else maxJumpMoves.add(jumpMoves.get(i));
        }
        return maxJumpMoves;
    }
 ...
```

Figure 3.7. updateJumpMoves(Board state, boolean currentTurn, ArrayList<> jumpMoves) updates the jump moves to take all consecutive jumps

If there is at least one generated jump move, the method will only return the jump moves. If there are no jump moves found, it will instead check for any regular moves. This is done similarly to the *getJumpMoves()* method; it looks for all of the current player's active pieces and checks if the forward (and backward for king) adjacent tiles are open.

```
...
   public ArrayList<CheckersMove> getLegalMoves(Board state, boolean currentTurn) {
        Tile[][] stateBoard = state.board;
        ArrayList<CheckersMove> legalMoves = getLegalJumpMoves(state, currentTurn);

        if (legalMoves.size() != 0)
            legalMoves = updateJumpMoves(state, currentTurn, legalMoves);

        else {
            for (int i = 0; i < ROW; i++) {
                for (int j = 0; j < COL; j++) {
                    if (stateBoard[i][j].isOccupied() && currentTurn == stateBoard[i][j].getPiece().isBlack()) {
                        if (stateBoard[i][j].getPiece().canMove(i - 1, j - 1, currentTurn))
                            legalMoves.add(new CheckersMove(i, j, i - 1, j - 1, false));
                        if (stateBoard[i][j].getPiece().canMove(i - 1, j + 1, currentTurn))
                            legalMoves.add(new CheckersMove(i, j, i - 1, j + 1, false));
                        if (stateBoard[i][j].getPiece().canMove(i + 1, j - 1, currentTurn))
                            legalMoves.add(new CheckersMove(i, j, i + 1, j - 1, false));
                        if (stateBoard[i][j].getPiece().canMove(i + 1, j + 1, currentTurn))
                            legalMoves.add(new CheckersMove(i, j, i + 1, j + 1, false));
                    }
                }
            }
        }

        return legalMoves;
    }
 ...
```

Figure 3.8. getLegalMoves(Board state, boolean currentTurn) generates the possible moves for the current player

- *checkWin(boolean currentTurn)* checks whether or not the current player has won. If the opponent player no longer has possible moves, then the current player wins. Otherwise, it will return false.

```
...
public boolean checkWin(boolean currentTurn) {
        return getLegalMoves(this, !currentTurn).size() == 0;
    }
...
```

Figure 3.9. checkWin(boolean currentTurn) checks if the current player has won

- *hasWinner()* checks whether the current board state is a game-over state. It checks if either player no longer has possible moves.

```
...
public boolean hasWinner() {
        return checkWin(false) || checkWin(true);
    }
...
```

Figure 3.10. hasWinner() checks if the current state is a terminal state

Generating the Game Tree

In order for the checkers agent to find the optimal strategy, it must choose the best possible move that can lead them to their win state, while also considering every possible response move by the player. To do this, the algorithm will generate a game tree, which is a graph that represents all possible board states within a game.

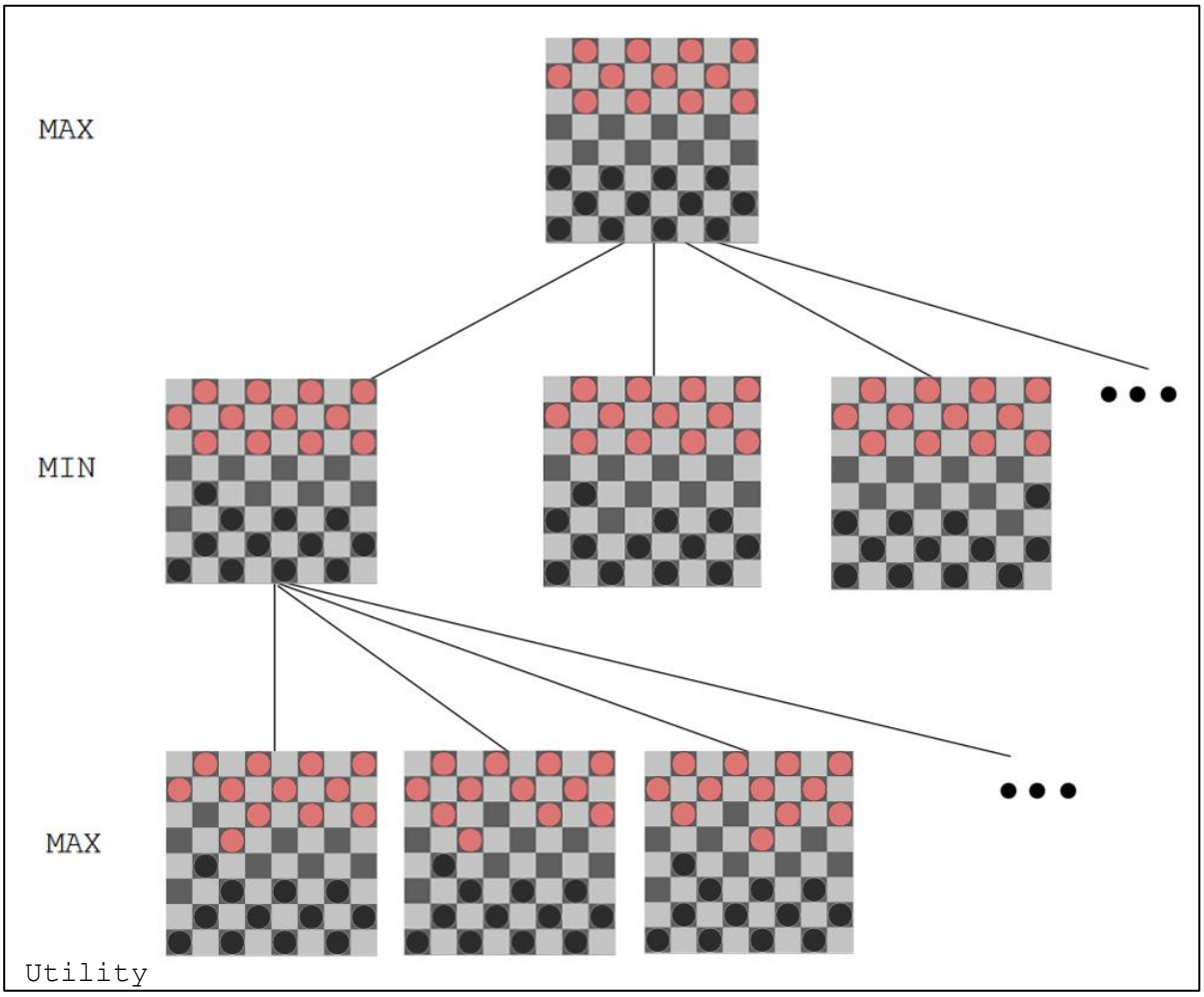


Figure 4. Example of a partial game tree for a checkers game

In the game tree in Figure 4, the root node is the current state of the board. The successor nodes represent decision states after a move, while the branches represent the decision moves. Each level switches to the other player. The players are represented as MIN and MAX. Each leaf node contains a utility value. MAX will try to move to a state of maximum value, while MIN will try to move to a state of minimum value. This game tree will be generated using the minimax algorithm.

```java
public CheckersMove miniMax(Board currentState) {
    ArrayList<CheckersMove> bestMoves = new ArrayList<>();
    double bestScore = Double.NEGATIVE_INFINITY;

    ArrayList<CheckersMove> legalMoves = currentState.getLegalMoves(currentState, true);

    if (legalMoves.size() == 0)
        return null;

    else if (legalMoves.size() == 1)
        return legalMoves.get(0);

    else {
        for (int i = 0; i < legalMoves.size(); i++) {
            CheckersMove checkedMove = legalMoves.get(i);
            Board tempBoard = new Board(currentState);
            tempBoard.getPiece(checkedMove.getOldRow(), checkedMove.getOldCol()).moveTo(checkedMove);
            double score = getMinValue(tempBoard, 1, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY);
            if (score > bestScore) {
                bestScore = score;
                bestMoves.clear();
                bestMoves.add(legalMoves.get(i));
            } else if (score == bestScore)
                bestMoves.add(legalMoves.get(i));
        }
    }
    // if there are more than one possible move with the best score, get a random move
    Random r = new Random();
    return bestMoves.get(r.nextInt(bestMoves.size()));
}
```

Figure 5.1. miniMax(Board currentState) returns the best possible move for MAX

The minimax algorithm in 5.1. returns the best possible move for the maximizer agent from the current board state, under the assumption that the opponent plays to minimize utility.

It will first generate all the possible moves for the agent. If the agent has no possible moves, it returns null and lead to a game over. If the agent has only one possible move, it will return that move since there is no other best move. Otherwise, it will generate the successor states from each possible move, and evaluate which move is the best by computing the minimax value of each move and choosing the move with the highest possible value. Since these successor states are at MIN's level, it will call the *getMinValue().* The minimax algorithm utilizes depth-first search, so it will explore as far as possible along each branch of moves before backtracking. If there is more than one move with the best maximum value, the algorithm will return a random move among the best possible moves.

```java
private double getMaxValue(Board currentState, int depth, double alpha, double beta) {

        if (depth == MAX_DEPTH || currentState.hasWinner()) {
            return currentState.getPointValue();
        }

        double maxEval = Double.NEGATIVE_INFINITY;
        ArrayList<CheckersMove> legalMoves = currentState.getLegalMoves(currentState, true);
        // for each legal move, simulate that move and get its minimax value
        for (int i = 0; i < legalMoves.size(); i++) {
            CheckersMove currentMove = legalMoves.get(i);
            Board tempBoard = new Board(currentState);
            tempBoard.getPiece(currentMove.getOldRow(), currentMove.getOldCol()).moveTo(currentMove);
            maxEval = Math.max(maxEval, getMinValue(tempBoard, depth+1, alpha, beta));
            // pruning
            if (maxEval >= beta)
                return maxEval;
            alpha = Math.max(maxEval, alpha);
        }
        return maxEval;
    }
```

```java
    private double getMinValue(Board currentState, int depth, double alpha, double beta) {

        if (depth == MAX_DEPTH || currentState.hasWinner()) {
            return currentState.getPointValue();
        }

        double minEval = Double.POSITIVE_INFINITY;
        ArrayList<CheckersMove> legalMoves = currentState.getLegalMoves(currentState, false);
        // for each legal move, simulate that move and get its point value
        for (int i = 0; i < legalMoves.size(); i++) {
            CheckersMove currentMove = legalMoves.get(i);
            Board tempBoard = new Board(currentState);
            tempBoard.getPiece(currentMove.getOldRow(), currentMove.getOldCol()).moveTo(currentMove);
            minEval = Math.min(minEval, getMaxValue(tempBoard, depth+1, alpha, beta));
            // pruning
            if (minEval <= alpha)
                return minEval;
            beta = Math.min(minEval, beta);
        }
        return minEval;
    }
```

Figure 5.2. getMaxValue(Board currentState, int depth, double alpha, double beta) and getMinValue(Board currentState, int depth, double alpha, double beta) generates the game tree all the way to the leaf nodes

The methods *getMaxValue()* and *getMinValue()* recursively call each other to generate the game tree all the way to the leaf nodes. If either method has not yet a reached a leaf node, it will generate the successor states of that node and call the other method to get their minimax values. Once either method reaches a leaf node, it will compute that board state's utility value and backtrack these values through the tree as the recursion unwinds. *getMaxValue()* returns the highest possible value for every MAX level, while *getMinValue()* returns the lowest possible value for every MIN level. Once the recursion backs up to the first level, the original *miniMax()* method will choose the move that had the highest minimax value, and return that move for the agent to execute.

The problem with the minimax algorithm is that for most games, generating a full game tree will be impossible due to its exponential growth. Furthermore, since the program does not have a checking condition for a draw state, the game can go on indefinitely if the player wants to. Therefore, the algorithm is set to have a maximum depth search limit of ten.

If the minimax algorithm has either found a win state or has reached the depth limit, it will measure the utility value of the evaluated state. This utility value determines how likely the current player is going to win based on the current board state. The computation for this utility value will be discussed in the next section.

The algorithm also utilizes alpha-beta pruning to reduce the number of states explored. If a state is proven to be unreachable if the opponent player plays optimally, then there is no need to explore its branches.

This pruning works by using the information gained from the explored branches. The highest value found for MAX is stored in alpha (α) and the lowest value found for MIN is stored in beta (β). If the evaluated node state's value makes α greater than or equal to β, then we can assume that this state will be avoided by the opponent. Therefore, the rest of the branches will no longer be explored. Otherwise, it will update the α-β values. In each MAX level, only α can be changed, while MIN can only change β.

Alpha-beta pruning will not affect the final result, but can significantly reduce the time taken to search for it.

	Without Pruning	With Pruning
Test 1	51.16 s	2.03 s
Test 2	52.16 s	1.99 s
Test 3	50.85 s	2.08 s
AVERAGE	**51.39 s**	**2.03 s**

Figure 6. Average execution time for the agent to search for the optimal decision from the initial state, with and without pruning. Depth limit is set to 10. Evaluated value is always -0.6.

Creating the Utility Function

The effectiveness of the minimax algorithm relies on the evaluation function of a given board state. The utility value is computed by the *getPointValue()* from the Board class. The utility value is computed using a weighted linear function, which is a summation of all the features of the board state multiplied by the weight associated with it.

The considered features of a board state are based on the checkers strategies on a 2021 article written by Howard Allen. These features along with their weights are listed below:

Evaluation Feature	Weight
Number of pawns	1
Number of kings	2
Number of pieces protecting the non-ideal back row positions	0.7
Number of pieces protecting the ideal back row positions	0.75
Number of pawns in the center of the board	0.4
Number of pawns in the center-edge of the board	0.2
Number of kings in the center of the board	0.8
Number of kings in the center-edge of the board	0.4
Number of pieces that are vulnerable	-0.5
Number of pieces that are protected	0.5

Figure 7. The features considered in computing the utility value of a given board state along with their weight values.

- Number of pawns (x1) – the number of pawn pieces each player has. The more pieces a player has, the more chances they have at winning.

- Number of kings (x2) – the number of king pieces each player has. Since a king piece has twice the movement opportunities compared to a pawn, its weight is doubled.

- Number of pieces protecting the non-ideal back row positions (x0.7) – protecting your king row is important to prevent your opponent from gaining a king piece. However, it is not ideal to not move all of your back row pieces since it reduces your chances to play offensively. The general strategy is to advance two of the four back row pieces.

- Number of pieces protecting the ideal back row positions (x0.75) – these are the two ideal back row positions that can successfully defend every square in front of them. It is ideal to leave these positions occupied by your piece for as long as you can.

- Number of pawns and kings in the center of the board – it is advantageous for the player to control the center squares as it provides more movement opportunities. For each piece in the center, its weight is incremented by 0.2 for every direction it can move into.

- Number of pieces that are vulnerable (x -0.5) – the number of pieces that the opponent can capture.

- Number of pieces that are protected (x 0.5) - the number of pieces that the opponent cannot capture.

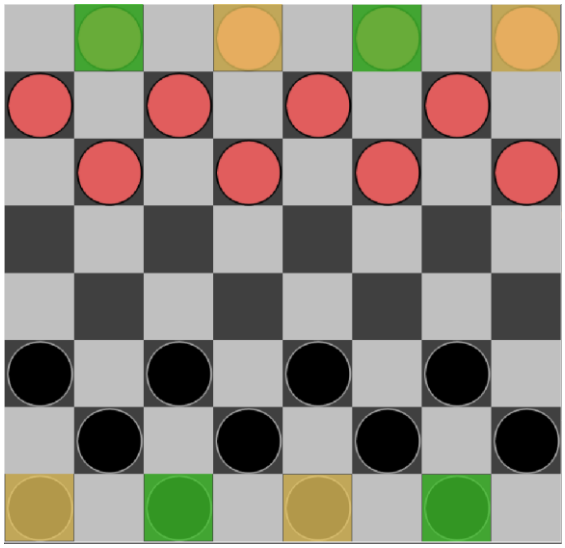


Figure 8. Green indicates the ideal back row pieces, while yellow indicates the non-ideal back row pieces.

To compute for the features' values, a nested for-loop was used to check every position on the board. If the current tile position contains a piece and has any of the features above, it will increment that feature's value in their respective index. Once all tiles are checked, it will subtract the human player's values from the agent player's values, then multiply them with their corresponding weight. The summation of these values will be the final utility value for the board state. The higher the value is, the more advantageous the evaluated board state is towards the agent.

```
/**
 * Features:
 * Index 0 - Number of pawns
 * Index 1 - Number of kings
 * Index 2 - Number of pieces protecting the non-ideal back row positions
 * Index 3 - Number of pieces protecting the ideal back row positions
 * Index 4 - Number of pawns in the center of the board
 * Index 5 - Number of pawns in the center-edge of the board
 * Index 6 - Number of kings in the center of the board
 * Index 7 - Number of kings in the center-edge of the board
 * Index 8 - Number of vulnerable pieces
 * Index 9 - Number of protected pieces
 * @return the utility value of this board state
 */

public double getPointValue() {
    int[] blackHeuristics = new int[10];
    int[] redHeuristics = new int[10];
```

```
for (int i = 0; i < ROW; i++) {
    for (int j = 0; j < COL; j++) {
        // Check every piece on the board
        if (board[i][j].isOccupied()) {
            Piece checkedPiece = board[i][j].getPiece();

            // check if pieces are protecting its king row
            if (checkedPiece.isBlack() && i == 7) {
                if (j == 2 || j == 6)
                    blackHeuristics[3]++;
                else blackHeuristics[2]++;
            } else if (!checkedPiece.isBlack() && i == 0) {
                if (j == 1 || j == 5)
                    redHeuristics[3]++;
                else redHeuristics[2]++;
            }

            // check if the piece is a king
            if (checkedPiece.isKing()) {
                if (checkedPiece.isBlack())
                    blackHeuristics[1]++;
                else redHeuristics[1]++;

                // check if the king piece is in the center
                if (i >= 2 && i <= 5) {
                    // center-edge
                    if (j == 0 || j == 7)
                        if (checkedPiece.isBlack())
                            blackHeuristics[7]++;
                        else redHeuristics[7]++;
                        // center-center
                    else if (checkedPiece.isBlack())
                        blackHeuristics[6]++;
                    else redHeuristics[6]++;
                }
            }
            // pawn pieces
            else {
                // check if the pawn piece is in the center
                if (i >= 2 && i <= 5) {
                    // center-edge
                    if (j == 0 || j == 7)
                        if (checkedPiece.isBlack())
                            blackHeuristics[5]++;
                        else redHeuristics[5]++;
                        // center-center
                    else if (checkedPiece.isBlack())
                        blackHeuristics[4]++;
                    else redHeuristics[4]++;
                }
            }

            // Check whether the black pieces can be eaten
            if (checkedPiece.isBlack() && (i > 0 && i < 7)) {
                // check non-edge pieces
                if (j > 0 && j < 7) {
                    // upper left
                    if (board[i - 1][j - 1].isOccupied() && !board[i - 1][j - 1].getPiece().isBlack()
                            && !board[i + 1][j + 1].isOccupied())
                        blackHeuristics[8]++;
                    // upper right
                    else if (board[i - 1][j + 1].isOccupied() && !board[i - 1][j + 1].getPiece().isBlack()
                            && !board[i + 1][j - 1].isOccupied())
                        blackHeuristics[8]++;
                    //lower left has king
                    else if (board[i + 1][j - 1].isOccupied() && (!board[i + 1][j - 1].getPiece().isBlack() &&
                            board[i + 1][j - 1].getPiece().isKing()) && !board[i - 1][j + 1].isOccupied())
                        blackHeuristics[8]++;
                    // lower right has king
                    else if (board[i + 1][j + 1].isOccupied() && (!board[i + 1][j + 1].getPiece().isBlack() &&
                            board[i + 1][j + 1].getPiece().isKing()) && !board[i - 1][j - 1].isOccupied())
                        blackHeuristics[8]++;
                }
            }
```

```
                // Check whether the red pieces can be eaten
                else if (!checkedPiece.isBlack() && (i > 0 && i < 7)) {
                    // check non-edge pieces
                    if (j > 0 && j < 7) {
                        // lower left
                        if (board[i + 1][j - 1].isOccupied() && !board[i + 1][j - 1].getPiece().isBlack()
                                && !board[i - 1][j + 1].isOccupied())
                            redHeuristics[8]++;
                        // lower right
                        else if (board[i + 1][j + 1].isOccupied() && !board[i + 1][j + 1].getPiece().isBlack()
                                && !board[i - 1][j - 1].isOccupied())
                            redHeuristics[8]++;
                        //upper left has king
                        else if (board[i - 1][j - 1].isOccupied() && (!board[i - 1][j - 1].getPiece().isBlack() &&
                                board[i - 1][j - 1].getPiece().isKing()) && !board[i + 1][j + 1].isOccupied())
                            redHeuristics[8]++;
                        // upper right has king
                        else if (board[i - 1][j + 1].isOccupied() && (!board[i - 1][j + 1].getPiece().isBlack() &&
                                board[i - 1][j + 1].getPiece().isKing()) && !board[i + 1][j - 1].isOccupied())
                            redHeuristics[8]++;
                    }
                }
            }
        }
    }
    // compute number of pawns
    blackHeuristics[0] = blackPieces.size() - blackHeuristics[1];
    redHeuristics[0] = redPieces.size() - redHeuristics[1];

    // compute number of protected pieces
    blackHeuristics[9] = blackPieces.size() - blackHeuristics[8];
    redHeuristics[9] = redPieces.size() - redHeuristics[8];

    double[] weights = {1, 2, 0.7, 0.75, 0.4, 0.2, 0.8, 0.4, -0.5, 0.5};
    double pointValue = 0;
    for (int i = 0; i < 10; i++)
        pointValue += (blackHeuristics[i] - redHeuristics[i]) * weights[i];

    return pointValue;
}
```

Figure 9. getPointValue() computes the utility value of the board state based on a set of features.

Testing the Algorithm

To test the minimax algorithm, the agent played against three opponents: a player who plays optimally, a player who is not playing optimally, and a player who plays optimally, but with a higher search depth limit.

For the first test case, both the player and the agent had to play optimally. This is done by giving the player a recommended move using the same minimax algorithm used by the agent. If the player plays every recommended move, then the game should eventually result in a draw. Surely enough, by Move 100, both the player and the agent had two king pieces remaining. By Move 150, both players have not been able to capture any more pieces and are simply avoiding each other's king pieces, making the game a draw. This means that both the minimax algorithm and the utility function is working as intended.

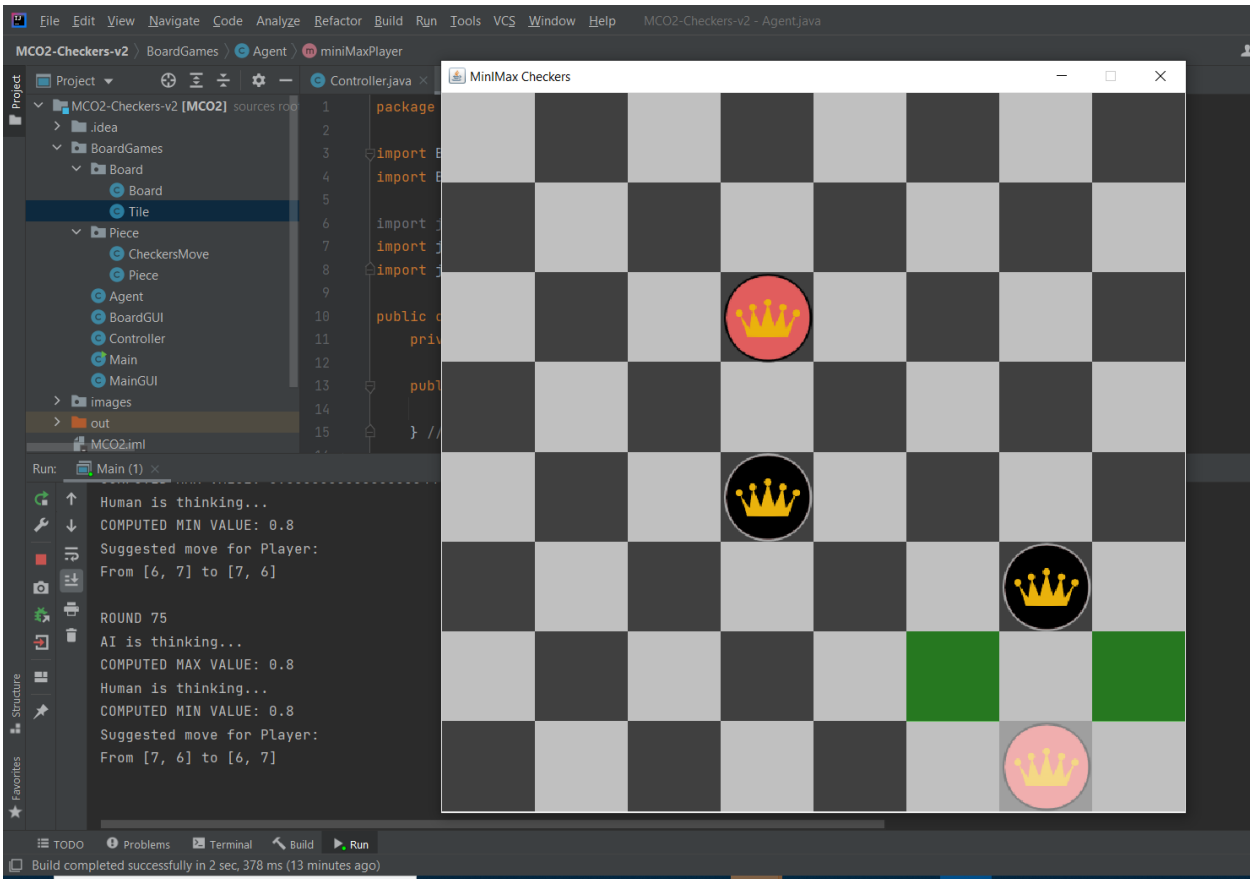


Figure 10.1. A Human vs. Agent game where both players are playing optimally. The console is providing the player their most optimal move. By the 150th move, both players are simply avoiding each other's king pieces, making the game a draw.

For the second test case, only the agent will play optimally. The human player will try to choose the best move, although they don't know the most optimal move for them. The agent won by Move 65. This means that the algorithm is able to effectively choose the best move for the agent, even if the opponent does not play optimally.

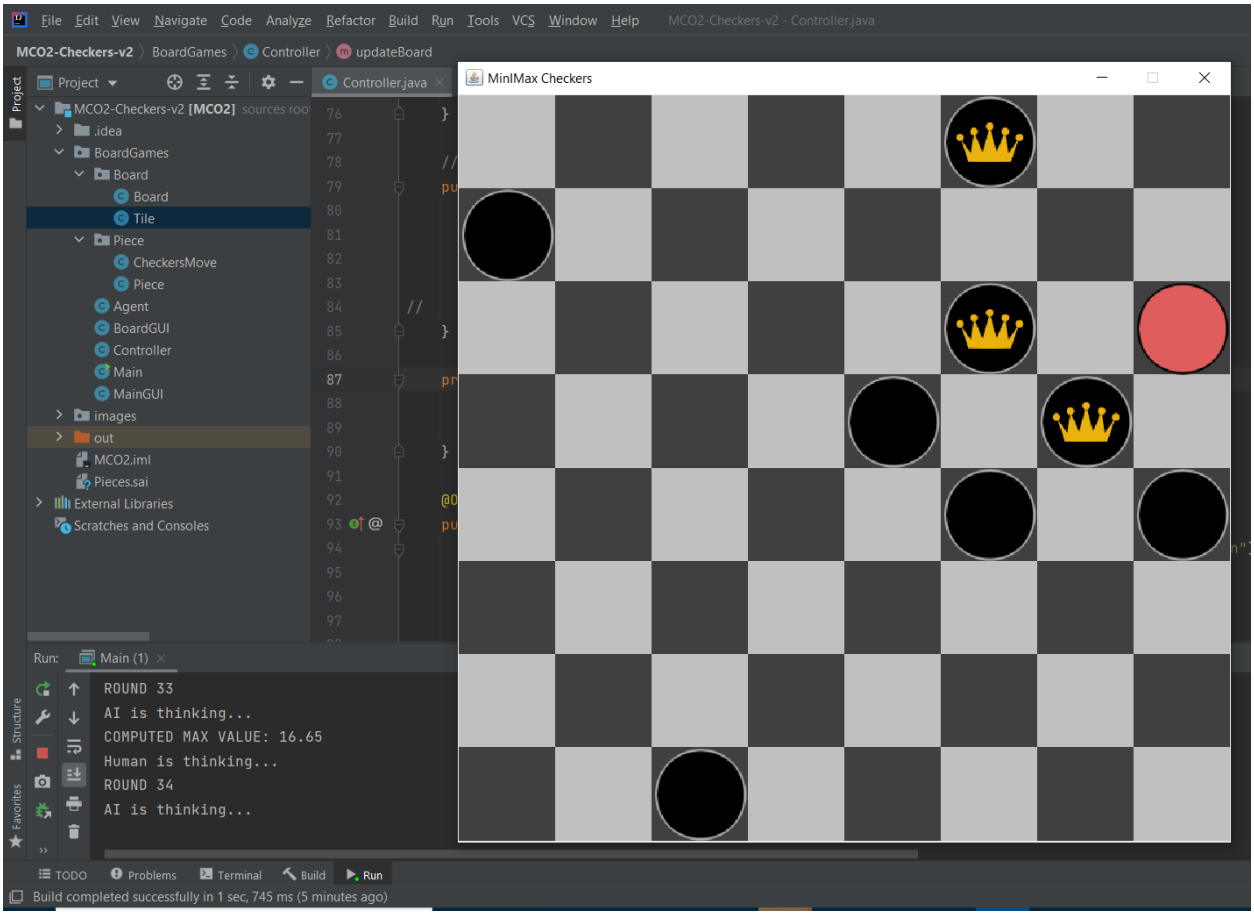
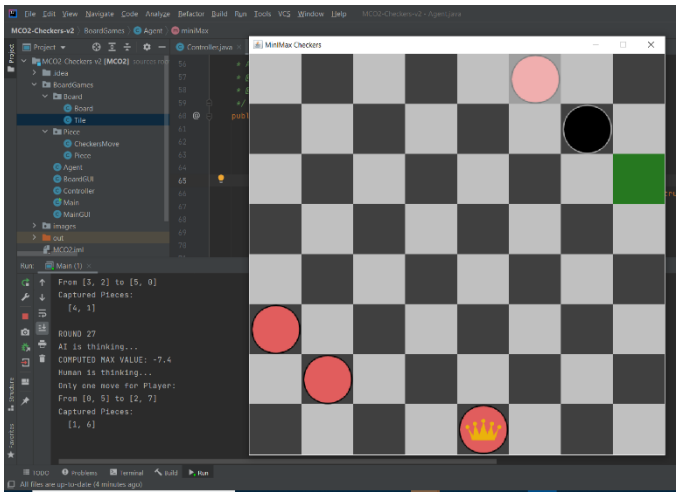
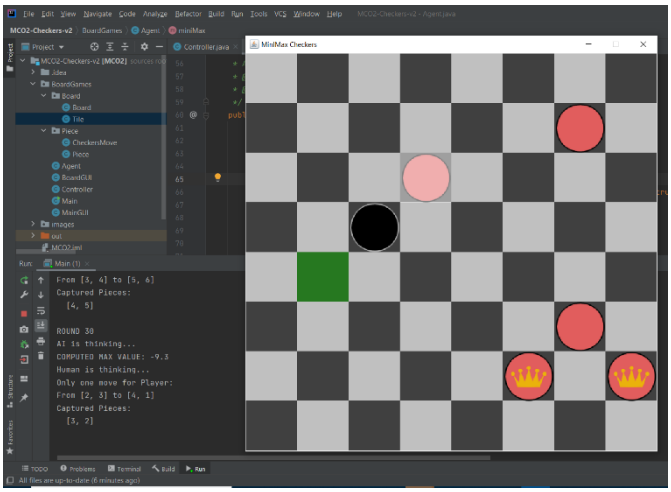


Figure 10.2. A Human vs. Agent game where only the agent plays optimally. By the 65th move, the agent has successfully blocked the human player from making any moves, giving the agent the win.

The last test case is similar to the first test case, only this time the agent's search limit is reduced to five. Even if both the human player and agent are playing optimally, and are using the same minimax algorithm and utility function, the human player always wins since their algorithm explores more levels compared to the agent. This proves that the depth of the game tree plays a significant role on how effective the minimax algorithm is.





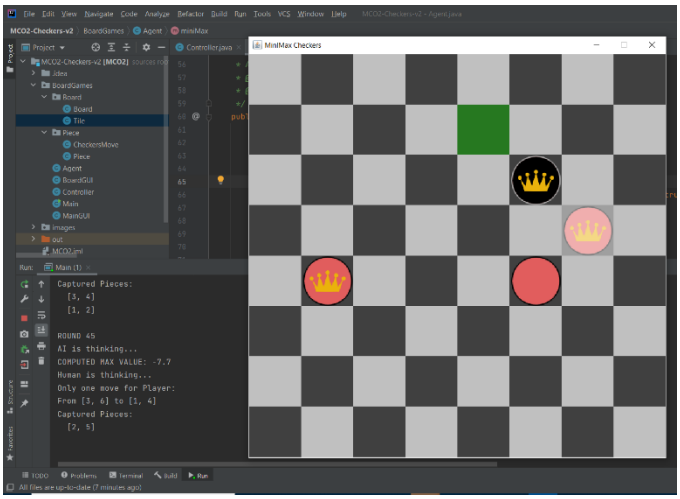


Figure 10.3. A Human vs. Agent game where both players play optimally. The player is given an optimal move, but the agent has a depth search limit of five and the human player has a depth search limit of ten. In all three test cases, the human player won.

Move Ordering

Good move ordering affects the effectiveness of alpha-beta pruning. For this program, the generation of successor states is entirely dependent on the *getLegalMoves()* method, which is shown in Figures 3.6, 3.7, and 3.8. It checks all possible moves in a procedural order.

Initially, the method checks the available moves for the current player by scanning the entire board in a top-down, left-to-right order. If it finds a piece belonging to the current player, it will check that piece's available moves in this order:

1. Check if the piece can make a jump towards top-right, top-left, bottom-left, and bottom-right.
2. Check if the piece can make a regular move towards the top-right, top-left, bottom-left, and bottom-right.

Since the program applies the forced jump rule, there is already some form of move ordering taking place. It will always generate the jump moves first. If the current player has at least one jump move, it will no longer generate any regular moves.

To measure the performance of the agent, the program will measure the time it took for the algorithm to find the optimal move. To measure the effectiveness of the move ordering function, it will take the ratio of the number of nodes explored per number of nodes generated. These computations will ignore scenarios when the agent has less than two possible moves.

To check if the move ordering function is optimal, both players will be playing optimally so that the game will always follow that same branch in each test case. The minimax algorithm using the initial move ordering function takes an average of 2.88 seconds to find the optimal move, and only explores around 60% of the total nodes generated. This means that on average, 3/5 successor nodes are explored per parent node.

The first test case checks all possible order combinations of directional moves (ex. check the forward moves first, check the backwards moves first, etc.). However, after the first few alternative order combinations, the difference between the ratios is too small to be significant. Therefore, it is assumed that any order of directional moves will result to the same performance.

Move Order	Average Number of Nodes Generated	Total Nodes Explored/Total Nodes Generated (%)	Average Execution Time (in seconds)
UL->UR->LL->LR Jump Moves -> UL->UR->LL->LR Regular Moves	2,031,879	63.30 %	2.88 s
LR->LL->UR->UL Jump Moves -> LR->LL->UR->UL Regular Moves	2,203,734	63.33 %	3.14 s
LL->LR-> UL->UR Jump Moves -> LL->LR-> UL->UR Regular Moves	2,048,032	63.19%	3.23 s

Figure 11. Average performance of the algorithm for the given order of moves in terms of nodes pruned and average execution time.

For the second test case, the moves are ordered by the number of pieces captured. Similar to the first test case, it does not significantly change the ratio of explored/pruned branches, but reduces the number of nodes generated.

Move Order	Average Number of Nodes Generated	Total Nodes Explored/Total Nodes Generated (%)	Average Execution Time (in seconds)
Original (Random) Move Ordering	2,031,879	63.30 %	2.88 s
Move Ordering by Number of Captured Pieces	2,020,021	63.19 %	3.49 s

Figure 12. Average performance of the algorithm with random move ordering vs. move ordering by number of captured pieces

For the third test case, the moves are sorted based on the evaluated utility function of their resulting state. This has significantly reduced the number of generated nodes and has effectively cut the number of explored nodes in half. However, since the utility function is associated with the board and not the move, it is costly to check the utility value of every possible move, since each move has to be generated a board state to be able to compute its utility. As a result, the execution time significantly increased.

Move Order	Average Number of Nodes Generated	Total Nodes Explored/Total Nodes Generated (%)	Average Execution Time (in seconds)
Original (Random) Move Ordering	2,031,879	63.30 %	2.88 s
Utility-Based Move Ordering	1,345,982	48.58 %	22.16 s

Figure 13. Average performance of the algorithm with random move ordering vs. utility-based move ordering

One recommendation to reduce the running time is to create a transposition table to store the utility value of each board state so that we don’t have to recompute it on subsequent occurrences. There might also be a way to evaluate the utility of the move itself without depending on its resulting state. This can be a weighted function of the number of captured pieces, the final position, if the piece cannot be captured by the opponent afterwards, etc.

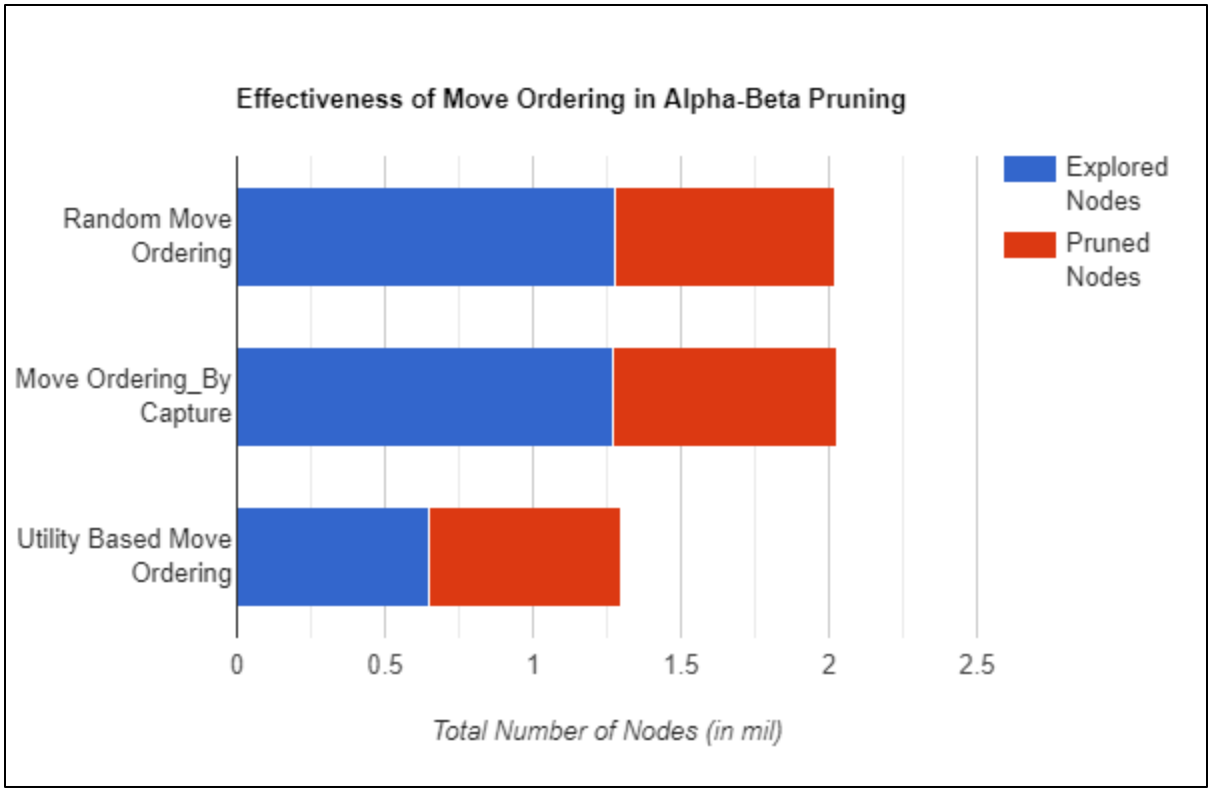


Figure 14. Comparison of the effectiveness of move ordering in pruning.

Figure 14 shows the average number of nodes explored and number of nodes pruned for each version of move ordering. There is no significant difference in ordering the moves based on direction or number of pieces captured. Ordering the moves based on their utility value can effectively reduce the search tree in half, although further improvements must be made to improve its running time. In comparison, the minimax algorithm without pruning generated 2.34 x 10^7 nodes from the initial state.

Closeness of Branching Factor to Perfect Move Ordering

From the test cases conducted for move ordering, the average number of legal moves a player has at a time is approximately 8. With a maximum depth of 10, the algorithm without pruning would check around 8^{10} or 1.07 x 10^9 nodes to find the best move. Thus, the ideal case of move ordering should only search $8^{10/2}$, or 32, 768 nodes.

The effective branching factor of the algorithm was computed by dividing the number of children nodes by the number of parent nodes. All test cases had an effective branching factor of 5. This means that the algorithm searches an average of 5^{10} or 9.77 x 10^6 nodes. If we include the forced jump moves into the average (turns that only generate one node), it reduces the branching factor to 4. It's not the ideal branching factor of $\sqrt{8}$, but still significantly reduces the number of nodes explored.

IV. **Reflection**

1. What are the skills I already possess which I used for this project?

   I already know how to create a two-player board game with a GUI thanks to our final machine project in CCPROG3, where we were tasked to create a board game called animal chess in Java. I also have a visual understanding about how the minimax algorithm and alpha-beta pruning works thanks to the lectures from this class. Having a good grasp on recursion has also helped me in creating the minimax algorithm.

2. What are the skills I had to learn for this project?

   Initially, I did not know how to duplicate an object in Java. I had to create my own version of the *deepcopy()* function in Python, since it was needed to create the successor states without altering the original board state. I also had to learn checkers strategies to create an effective utility function. Finally, I had to learn how to modify a two-player game into a player vs. AI game, where the program makes its move as soon as the player makes a valid move.

3. What are the challenges I encountered and how did I solve it? (may be technical, or contextual such as poor internet connection, machine is too slow, etc.)

   Generating the legal moves was the hardest part for me, particularly the jump moves. I implemented the force-jump rule, including any consecutive jumps. Since the code that checks if a piece can initially jump is hard-coded to check certain positions, it was difficult for me to keep track of consecutive jumps. To solve this, I applied the concepts I learned from MCO1 about backtracking. Instead of creating a new canJump function, I opted to make a recursive function that simulates each jump move in a temporary board state, then update the original jump moves once all consecutive jump moves are found.

References:

Allen, H. (2021). *Checkers strategy and tactics: How to win every time.* HobbyLark. Retrieved from https://hobbylark.com/board-games/Checkers-Strategy-Tactics-How-To-Win

Checkers: Standard rules. (2016). LearnPlayWin. Retrieved from https://learnplaywin.net/checkers-rules/