IMPLEMENTING GAME-TREE SEARCHING STRATEGIES
TO THE GAME OF HASAMI SHOGI

A graduate project submitted in partial fulfillment of the requirements
for the degree of Master of Science
in Computer Science

By

Pallavi Gulati

December 2012
The graduate project of Pallavi Gulati is approved by:

__________________________________________  ________________________________________
Peter N Gabrovsky, Ph.D                        Date

__________________________________________  ________________________________________
John J Noga, Ph.D                              Date

__________________________________________  ________________________________________
Richard J Lorentz, Ph.D, Chair                 Date

California State University, Northridge
ACKNOWLEDGEMENTS

I would like to acknowledge the guidance provided by and the immense patience of Prof. Richard Lorentz throughout this project. His continued help and advice made it possible for me to successfully complete this project.
# TABLE OF CONTENTS

**SIGNATURE PAGE** ......................................................................................................................... ii  
**ACKNOWLEDGEMENTS** ...................................................................................................................... iii  
**LIST OF TABLES** ................................................................................................................................. v  
**LIST OF FIGURES** ................................................................................................................................. vi  
1 **INTRODUCTION** ................................................................................................................................. 1  
1.1 The Game of Hasami Shogi ............................................................................................................... 1  
1.2 The program HSQueenBot .............................................................................................................. 2  
1.3 Problem Statement & Research Questions ...................................................................................... 6  
2 **RANDOM ALGORITHM** ......................................................................................................................... 7  
2.1 Random Algorithm as applied to Hasami Shogi .............................................................................. 7  
2.2 Issues with Random on Hasami Shogi and resolution ..................................................................... 7  
2.3 Evaluation Function ........................................................................................................................... 8  
3 **MINI-MAX AND ALPHA-BETA PRUNING ALGORITHMS** .......................................................... 9  
3.1 Mini-Max Algorithm ........................................................................................................................... 9  
3.2 Mini-Max as applied to Hasami Shogi ............................................................................................... 11  
3.3 Issues with Mini-Max on Hasami Shogi and resolution .................................................................. 11  
3.4 Alpha-Beta Pruning Algorithm ........................................................................................................ 12  
3.5 Alpha-Beta Pruning as applied to Hasami Shogi ............................................................................ 13  
3.6 Issues with Alpha-Beta Pruning on Hasami Shogi and resolution ................................................. 14  
4 **MONTE-CARLO TREE SEARCHING (MCTS) AND UPPER CONFIDENCE BOUNDS (UCT) ALGORITHM** .................................................................................................................. 15  
4.1 Monte-Carlo Tree Searching (MCTS) Algorithm ............................................................................ 15  
4.2 MCTS as applied to Hasami Shogi .................................................................................................... 17  
4.3 Issues with MCTS on Hasami Shogi and resolution ....................................................................... 19  
4.4 Upper Confidence Bounds (UCT) Algorithm .................................................................................. 20  
4.5 UCT as applied to Hasami Shogi ...................................................................................................... 21  
4.6 Issues with UCT on Hasami Shogi and resolution ......................................................................... 22  
5 **TEST RESULTS** ................................................................................................................................. 23  
5.1 Development and Test Environment ............................................................................................... 23  
5.2 Results ................................................................................................................................................ 23  
6 **CONCLUSIONS AND FUTURE RESEARCH** ............................................................................... 26  
6.1 Answers to Research Questions ....................................................................................................... 26  
6.2 Conclusions and observations ......................................................................................................... 26  
6.3 Future Research and Development ................................................................................................. 27  
**REFERENCES** ...................................................................................................................................... 28  
**APPENDIX** .......................................................................................................................................... 30
LIST OF TABLES

Table 1: Results of Random Algorithm against all the rest.................................................. 24
Table 2: Results of Min-Max Algorithm vs Alpha-Beta Algorithm....................................... 24
Table 3: Results of MCTS Algorithm vs UCT Algorithm ...................................................... 24
Table 4: Results of Mini-Max with Alpha-Beta against MCTS with UCT.............................. 25
Table 5: Results of Mini-Max with Alpha-Beta against MCTS with UCT.............................. 25


<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mini-Max Algorithm example [18]</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Alpha-Beta Pruning example from [7]</td>
<td>12</td>
</tr>
</tbody>
</table>
The main purpose of this project was to understand how the various gaming algorithms apply to the game of Hasami Shogi. The game of Hasami Shogi posed its own challenges as the number of possible moves from any particular state of the board is quite large. Different algorithms like Mini-Max, Alpha-Beta, Monte-Carlo Tree Search were implemented and tried out. Monte Carlo Tree Search with UCT method seemed to be more effective than the others though not very significantly. The UCT value needed to be optimized for the game and also some evaluation mechanisms needed to be put in place so that the program would concentrate on a particular promising part of the game-tree instead of wasting time evaluating moves which cannot lead to a winning scenario. The program provides an option to select different algorithms for both players and a comparison study was made. The results of all these games have been published as part of the thesis.
1 INTRODUCTION

1.1 The Game of Hasami Shogi

Hasami Shogi is a traditional Japanese game played on a shogi-ban, i.e. a 9x9 grid of spaces, using the pawns from the shogi set. The black and white pieces (18 pieces in total) initially occupy the entire back row on either side, i.e. the row of spaces nearest to the controlling player.

MOVE – A piece moves like a rook in chess, that is, it can move any number of empty squares along a straight line in any orthogonal direction, limited only by the edge of the board. A piece cannot jump over other pieces nor can it move diagonally. A player cannot pass a move.

CAPTURING – The word “Hasami” means sandwiching. An opponent’s piece is taken by custodian capture which is capture by sandwiching opponent’s pieces horizontally or vertically (but not diagonally) between a player’s own pieces. A piece must be moved into such a position immediately on either side of an opponent’s piece to make a capture. If a piece is moved between two of the opponent’s pieces, it is not captured.

Custodian capture is a technical term in board games referring to a particular form of capturing. It occurs when a player has placed two of his pieces on opposite sides of an opponent's piece. This mode of capture is unusual in most modern games and was most popular during the Dark Ages, particularly in Northern Europe [2].
Multiple captures can be made simultaneously. This occurs only if a piece is moved so that it causes more than one of the opponent’s pieces to be positioned between the piece moved and another of the player’s own pieces.

GOAL – The object of the game is to be the first player to capture 8 out of 9 of an opponent’s pieces.

There are some variants to the basic Hasami Shogi game, for example Dai Hasami Shogi with 18 pieces for each player, Diagonal Capture allowed, Single piece capture only, War Variant where you align 5 of your pieces diagonally, etc. and generally rules are pre-agreed on before beginning the game.

1.2 The program HSQueenBot

A computer program was developed named HSQueenBot which implemented the game of Hasami Shogi using the different game-playing algorithms. The different game-playing algorithms that were implemented for the purpose of this research are:

- Random
- Mini-Max
- Mini-Max with Alpha-Beta Pruning
- Monte-Carlo Tree Search (MCTS)
- Monte-Carlo Tree Search with Upper Confidence Bounds (UCT)

The structure of the code is inherently object-oriented by dividing into separate classes for board, tree, AI and program. The BOARD class handles all functions related to the board like playing the moves, getting legal moves, deciding the winner, etc.
The PROGRAM class is the main class which does the game playing. The algorithms to be used for the WHITE player and the BLACK player need to be set in this class and on execution the moves will be generated and played accordingly. The board, moves and results are displayed by the program class on to the command screen along with the end result of the game i.e. whether the WHITE player or the BLACK player won.

The TREE class and NODE class handle the adding of the child nodes for simulation, managing the game pointers and simulation pointer.
The main intelligence resides in the AI class which has independent functions for each of the game-playing algorithms. The complete logic of the algorithms has been implemented in their individual functions with just the evaluation of the moves being done through a separate function in the same AI class.
A GUI interface was partially designed which would offer the ability to play against human players but it was adding unnecessary runtime and complexity to the code and hence it was eliminated from the final version of the code. But adding an interactive GUI has been identified as a future enhancement to this project. Currently the bot prints the board on the command screen after every move.

Sample Output Prompt:
1.3 Problem Statement & Research Questions

Which gaming algorithm seems to work best when applied to the game of Hasami Shogi?

Does the Monte Carlo Tree Search fare best for the game of Hasami Shogi just like it does for many other games like Poker, Scrabble, Go, Settlers of Catan [16]?

Can the algorithms be implemented without much of the inherent knowledge of the game of Hasami Shogi or do you need to implement an evaluation function to decide moves?

Are there any improvements that can be made to the various algorithms to make it work better for Hasami Shogi?
2 RANDOM ALGORITHM

2.1 Random Algorithm as applied to Hasami Shogi

A random algorithm as the name suggests was a function which played a random move and was developed mainly to be used to see how the other game-playing algorithms perform in comparison. HSQueenBot applied the Random function to Hasami Shogi by first determining all the possible legal moves for a particular board layout and then picking a random move from the list and playing it.

2.2 Issues with Random on Hasami Shogi and resolution

The issue seen with playing randomly other than the obvious was that the game did not converge and the game went on endlessly with both sides sometimes playing the similar kind of moves over and over again. The number of possible legal moves from any board layout could be anywhere between 60-150 moves initially when most of the pieces are on the board. Most of these moves do not lead to capturing of pieces and could not possibly lead to a win condition. So the random function needed some kind of evaluation function to filter out the unnecessary moves based on some criteria and then have the random function pick one random move from the remaining good moves.

This evaluation function was also to be used during MCTS random playouts to ensure that the game ended and results could be back-propagated in a certain time window and not have a random playout of unintelligent moves which don’t go anywhere.
2.3 Evaluation Function

A move evaluation function was added which would assign a rank to each of the possible legal moves for a particular board position and player, based on how many of the opponent’s pieces could potentially be captured if that move is picked and played. For each piece captured 10 points were added to the RANK of the move. Also when a capture was not the result but if you were able to at least place one of your pieces next to an opponent’s (with a possible capture one move later) it was assigned a RANK of 2 points just to make it a little more preferential over moves which are locations on the board in the middle of nowhere.

This seemed to work very well with the random function as well as the other algorithms where the same evaluation function was re-used. The only issue observed with the evaluation function was the fact that it looked only one move ahead and that it was mainly an offensive-strategy based evaluation looking for captures. There was no defensive-strategy added to it. So the evaluation function would not try to save equal or more of its own pieces instead of capturing an opponent piece. This would sometimes lead to games where algorithms would do nothing but capture each other’s pieces one after the other and clear the board. But adding a defensive-strategy to the existing code needed it to see two moves ahead and the current code would need some major changes to handle that. Currently everything is based on the list of the next possible moves and having a next-next move evaluated for every next move would need a major re-structure of the code. Also how much benefit could be obtained by doing this is questionable. This was not implemented as part of this project but can be a good candidate for future enhancements.
3 MINI-MAX AND ALPHA-BETA PRUNING ALGORITHMS

3.1 Mini-Max Algorithm

The mini-max algorithm is a recursive algorithm for choosing the next move in a two-player game. An evaluation function is implemented which computes the “goodness” of a position based on pre-defined criteria. Basing the evaluation function as a weighted sum of the factors influencing it is a widely used strategy [17].

The mini-max algorithm can be thought of as involving two players – the maximizing player and the minimizing player. A depth-first search-tree is generated from the current game position until the end of the game. This end of the game position is evaluated and an evaluation value is determined based on specific game-based criteria which could be opponent pieces captured, own pieces lost, win/loss/draw condition, etc. The inner nodes of the tree are then filled bottom-up with the evaluated value. The nodes corresponding to moves of the maximizing player receive the maximum of the evaluated values of all its children nodes. The nodes of the minimizing player will select the minimum value of its children nodes [9].

Figure 1 below shows a Mini-Max Algorithm example from [18]. The evaluation function in this example assigns values of 2 and 3 to the left-most sub-tree. The parent of these child nodes is the maximizing player and hence the evaluation value propagated up is the maximum of 2 and 3 which is 3. Further up is the parent node of these maximizing nodes with evaluation values 3 and 9. This is a minimizing node and so the value assigned to
this node is the minimum of 3 and 9 which is 3. The same is repeated by all sub-trees till they reach the root.

Figure 1: Mini-Max Algorithm example [18]

Given the rules of any two-person game with a finite number of positions, one can always trivially construct a Mini-Max algorithm that would exhaustively traverse the game tree. However for many non-trivial games such an algorithm would require an infeasible amount of time to generate a move in a given position [5].

Generating the whole game-tree for even simple games would take a very long time and for most games like chess which have a big branching factor, it might not even fit into memory. So optimizations need to be added to the algorithm. One basic optimization is to limit the depth of the search tree. Another solution is to limit the time for search and choose the best move found until the moment the time runs out [9].
3.2 Mini-Max as applied to Hasami Shogi

HSQueenBot implemented the Mini-Max Algorithm recursively to pick the best move for the player by assigning a rank to each move. Initially the criterion used was whether the current player loses or wins when playing the particular move. The algorithm would then apply a RANK of +1 when BLACK wins and -1 when WHITE wins.

3.3 Issues with Mini-Max on Hasami Shogi and resolution

Just using the final win/loss for evaluation did not seem to work well when sometimes none of the legal moves could lead to a win or loss. There seemed to be games when the assigned RANK for all the legal moves would be 0 and then one of the moves would get picked i.e. the first one evaluated since it would not subsequently find a better move to replace the default assignment. In most of these cases, there would be an obvious move (obvious to the human eye) where an opponent’s piece could have been captured and that should have been the move picked.

To take care of this issue, the evaluation was then based on the number of pieces each of the legal moves would capture rather than just win/loss. A number based on this was to be assigned as RANK and since there are at most 9 opponent pieces each was assigned a weight of 10 with +/- 100 being the value used for assigning a WIN/LOSS to the RANK.

Another issue was the amount of time needed to go all the way to the end of a game to register a win/loss. In most cases, the algorithm needed to be terminated after some time and hence depth of the search needed to be limited so that function could return in a
reasonable time frame. So the variable of depth was added to stop the recursive call and various values for this depth factor were tried out.

3.4 Alpha-Beta Pruning Algorithm

This is a search optimization algorithm that reduces the number of nodes that need to be evaluated in the search tree by the Mini-Max Algorithm. This algorithm optimizes the search by comparing the current move evaluation with the other moves evaluated before. If the current move is in a worse position compared to earlier ones then it does not make much sense to evaluate it any further and time can be saved by discarding it.

For example [7], Node K holds a value of 2. This value is smaller than what node C has, 6. This means that nodes L and M cannot possible influence node A and are subsequently pruned. This is because D is a minimizing node, and L and M would not change D unless they were smaller than 2. But D will not influence A unless K had been a number larger than 6.

![Figure 2: Alpha-Beta Pruning example from [7]](image-url)
The way pruning works is that for any node $n$, if a player has a better move at the parent of $n$ or at any move further up, then the node $n$ will never be reached in actual play. So when enough information about $n$ is determined by examining its predecessors, it gets pruned.

This technique of discarding moves does not in any way affect the final result and at the same time speeds up the search algorithm. The main advantage of this optimization is that whole branches of the tree can be eliminated. The search time saved can be then utilized to increase the look ahead or for a deeper search on a more promising move.

### 3.5 Alpha-Beta Pruning as applied to Hasami Shogi

HSQueenBot implemented the Alpha-Beta Pruning algorithm by maintaining two values, alpha and beta which represent the maximum score that the maximizing player is assured of and the minimum score that the minimizing player is assured of respectively. Initially alpha is assigned a large negative number whereas beta is a large positive number.

This window or gap between the two values becomes smaller and smaller as the recursion progresses. When the beta value becomes less than the alpha value, it means that the current position need not be explored further as it cannot produce a favorable result. These child nodes are pruned. The pseudocode for implementation of the Alpha-Beta Pruning algorithm looks something like the below-
3.6 Issues with Alpha-Beta Pruning on Hasami Shogi and resolution

The main issue on implementing Alpha-Beta Pruning was ascertaining if the algorithm is generating an identical move as Mini-Max Algorithm. Also another issue was whether any of the moves were really getting pruned or not and if so were they really helping in any significant manner. Based on the results of the test runs it did not seem to make any significant difference whether the bot was playing plain vanilla Mini-Max or the one with Alpha-Beta Pruning.
4 MONTE-CARLO TREE SEARCHING (MCTS) AND UPPER CONFIDENCE BOUNDS (UCT) ALGORITHM

4.1 Monte-Carlo Tree Searching (MCTS) Algorithm

Monte-Carlo Searching is another game-playing mechanism that can be used to determine the most promising next move. It is a sampling-based approach, which collects many samples to approximate the outcome of each move. A sample is a random game in which both players randomly select a legal move from a list of possible legal moves until the final score of that random game is determined. This is iteratively done until the game is over. Then based on which legal move won most of its random playouts, the actual next move to be played is decided.

Monte-Carlo Tree Search (MCTS) can be thought of as an extension to Monte Carlo Searching based on randomized explorations of the search space. Using the results of previous explorations, the algorithm gradually grows a game tree in memory. This game-tree remembers all the random playouts since the start of the game and every time new random playouts get added to the tree or if it exists already, the results are incremented. Thus as the game progresses each possible move has more and more previously simulated playouts and so it successively becomes better at accurately picking the most promising move.
MCTS comprises four strategic phases which are repeated over and over based on either a count of simulations or elapsed time. These four phases are:

- **Selection** – In the selection phase the search tree is traversed from the root node until it selects a leaf node that is not simulated yet.
- **Expansion** – In the expansion phase, the leaf node is added to the tree.
- **Simulation** – In this phase the game is simulated to play randomly from the newly added child node until the end of the game, resulting in either a win/loss or a draw.
- **Back-propagation** – The end-game results are propagated up the tree.

The move played by the program is the child of the root with the highest visit count [12]. It would seem that the child node with the highest WINS / VISITS ratio should be picked. But that is not the right choice since the wins registered could just be because of some lucky random moves in a particular playout which lead to a win. In case more simulations were run on the same node, a different result would emerge. Also the ratio of WINS / VISITS could be high merely because the number of visits of that node is not very high as compared to other nodes.

But a child node with a high visit count would imply that it was found to be of interest repeatedly by the selection phase and picked for simulation playouts. Thus at the end of the simulation, the move picked to be played is the child node with the highest visit count.
An outline of the four steps of MCTS is depicted below in Figure 3:

![Diagram of MCTS algorithm]

**Figure 3: Four phases of MCTS algorithm from Chaslot (2006) [11]**

### 4.2 MCTS as applied to Hasami Shogi

The program HSQueenBot implements the MCTS algorithm in a function which allows it to be used from the main program class. The function builds a search tree which gets populated over time as more and more random playouts are made. In the selection phase, the algorithm looks for all the possible legal moves from the current root of the tree. It selects the legal move with the highest rank which is not part of the tree yet (i.e. it has not been simulated yet). Once selected the move gets added to the tree as part of the expansion phase.
Pseudocode for MCTS Phases:

```csharp
// EXPANSION PHASE - Add all the possible legal moves from this position to tree
List<Move> listPossibleMoves = new List<Move>();
listPossibleMoves = b.PossibleMoves(p);

if (listPossibleMoves != null)
{
    // Add evaluation to somehow identify more promising moves amongst them
    // Evaluation Function
    listPossibleMoves = MoveEvaluation(b, p, listPossibleMoves);

    for (int m = 0; m < listPossibleMoves.Count; m++)
    {
        t.AddLeafNode(t.gamePtr, listPossibleMoves[m], p);
    }
}

// SELECTION PHASE - Pick one of the unexplored nodes / exploit existing nodes
Node[] currentNodes = t.GetAllChildNodes(t.gamePtr);
for (int q = 0; q < currentNodes.Length; q++)
{
    if (currentNodes[q].move.rank > maxRank)
    {
        i = q;
    }
}

if (bestChild == null || bestChild.numVisits == 0 ||
    currentNodes[i].numWins / currentNodes[i].numVisits >
    bestChild.numWins / bestChild.numVisits)
{
    bestChild = currentNodes[i];
    ...
```

A multiple number of random playouts are played in the simulation phase from this child node and the win/loss results are propagated and stored up the tree in the back-propagation phase. Each node stores two values – the number of VISITS and the number of WINS. The number of visits is the number of times any node (each node represents a move) was reached or picked for random simulations so far in the game. The number of wins represents the number of times the current player won at the end of the simulated game where this node or move was a part.
The simulation pointer is used to backtrack all the way up the tree and the number of visits of each node along the path is incremented but the number of wins of only those nodes which match the player which won the simulation playout are incremented.

**Pseudocode for MCTS Phases:**

```java
// SIMULATION PHASE - Simulate a random playout on copyOfB board
Board copyOfb = new Board();
copyOfb.Clone(b);
copyOfb.PlayThisMove(currentNodes[i].move);
// Assign the simulationPtr to the selected move
t.simulationPtr = currentNodes[i];
...
if (t.simulationPtr.isNodeBlackWhite == copyOfb.DidAnyoneWin())
{
    // BACKPROPAGATION PHASE
    t.simulationPtr.numWins++;
}
// BACKPROPAGATION PHASE
    t.simulationPtr.numVisits++;
t.simulationPtr = t.simulationPtr.parent;
```

The four phases are repeated over and over until time permits. At the end of the timer, the move to be played is picked which is the child node with the maximum number of VISITS.

### 4.3 Issues with MCTS on Hasami Shogi and resolution

Initially on implementing MCTS the issues faced were similar to the random algorithm where they seem to be not picking obvious intelligent moves. And also the games would not terminate within a reasonable time. Another issue was that as the number of possible
legal moves is very large, the child nodes picked to be simulated were not very promising to begin with. Hence a small modification to the basic code was made where the legal moves were shortlisted based on the fact that if the current player’s piece can be move to already existing black-next-to-white pieces then the probability of capture and subsequent win would be higher. The existing evaluation function was modified and utilized.

4.4 Upper Confidence Bounds (UCT) Algorithm

Upper Confidence Bounds applied to trees is an extension which can be applied to Monte-Carlo Searching. This algorithm suggests that during the selection phase of MCTS the node to be selected should be the one with the maximum value of a certain formula.

An Upper Confidence Bounds (UCB) formula of the following form [3] is typically used:

$$ v_i + C \times \sqrt{\frac{\ln N}{n_i}} $$

where $v_i$ is the estimated value of the node i which is the wins/visits value,
$n_i$ is the number of times the node i has been visited,
$N$ is the total number of times that its parent has been visited,
$C$ is a tunable bias parameter.

The UCB formula balances the exploitation of known promising nodes with the exploration of unvisited nodes. The first term is the wins to visits ratio of unexplored nodes and is what MCTS uses. MCTS is the case when the second term in the above equation is zero i.e. the tunable bias parameter $C = 0$. 
The second factor in the equation is the exploitation bias where importance is assigned to the fact that the parent of the node has been visited a certain times. The more times the parent is visited would indirectly mean that the child nodes further down are worth exploring further and should be picked by the algorithm. As the number of times a particular node is visited starts to go up, the UCT estimates start becoming more reliable i.e. the estimates will typically be unreliable at the start of the search.

4.5 UCT as applied to Hasami Shogi

HSQueenBot implemented the UCT algorithm by starting at the current root which is the node pointed to by the game pointer and finding a path to a child node with the highest UCB value i.e. choose a child node which has the highest exploration / exploitation value. This is the sum of the exploitation term (the percentage of won simulations from this child node) and the exploration term (the square root of the ratio log(number simulations of the parent) / (number simulations of this child). Then randomly play from the child node until the game is over. At the end of the game, update the statistics i.e. number of simulations in each node and the number of wins for black / white up the tree. If there is time, go back to the root and repeat the process.

As with the MCTS algorithm, the UCT algorithm also returns the best move to be played by the game as the child node which is the most simulated node from the root [12] i.e. the one with the maximum visit count.
Pseudocode for UCT Algorithm:

```java
public Move UpperConfidenceBoundsMove(Board b, Player p, Tree t) {
    ...
    double C = 2;


    if (currentNodeUCTvalue > bestChildUCTvalue) {
        bestChild = currentNodes[i];
    }
    ...
}
```

4.6 Issues with UCT on Hasami Shogi and resolution

The most important challenge when implementing UCT was determining the value of the tunable bias parameter C in the above UCB formula. This value has to be determined experimentally by observing changes in the results for different values of C which at times were difficult to gauge. Values of 1, 2 and √2 were tried out. A limited number of tests were run with values of C = 10 and 100.

Another issue was for UCT to make any difference over MCTS, the parent nodes needs to have many number of simulations done prior to evaluation. As the number of simulations done on the parent as well the child nodes start becoming very high, the most promising move picked by the algorithm gets more accurate.
5 TEST RESULTS

5.1 Development and Test Environment

The project was implemented on the Microsoft .net 2.0 platform using C# running on a Windows 7 machine with 6GB RAM and Intel i7 processor. The program provided an option for selecting the different strategies for evaluation. I could not find any game-hosting websites which allows hooking up a Hasami Shogi playing bot nor any standard protocol to be used to communicate with other similar programs. All of the evaluations were based on a comparison study of the algorithms implemented.

The various algorithms to be played by the WHITE and BLACK players were selected through the main program and on execution the game would play continuously and print the board for every move and the final result as well.

5.2 Results

The results of each of the algorithms against the rest are tabulated below. The Random Algorithm as expected did not do very well as compared to the game-playing algorithms and roughly won 5% of the games played. The results were slightly better when playing first and using the same evaluation function.
Table 1: Results of Random Algorithm against all the rest

<table>
<thead>
<tr>
<th>Algorithm Random vs</th>
<th>Total Games played</th>
<th>Random Wins</th>
<th>Random Win %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mini-Max</td>
<td>145</td>
<td>7</td>
<td>4.8%</td>
</tr>
<tr>
<td>Alpha-Beta</td>
<td>100</td>
<td>5</td>
<td>5.0%</td>
</tr>
<tr>
<td>MCTS</td>
<td>125</td>
<td>8</td>
<td>6.4%</td>
</tr>
<tr>
<td>UCT</td>
<td>75</td>
<td>4</td>
<td>5.3%</td>
</tr>
</tbody>
</table>

The comparison study between the Mini-Max algorithm and the Alpha-Beta Pruning algorithm when run against each other did not show any one to be dominant over the other. Thereby implying that the Alpha-Beta pruning algorithm does not really differ as far as the most-promising move returned it only does that faster than the Mini-Max algorithm. Similar results were observed when MCTS was pitted against UCT. Neither one was dominant over the other.

Table 2: Results of Min-Max Algorithm vs Alpha-Beta Algorithm

<table>
<thead>
<tr>
<th>Algorithm Mini-Max vs</th>
<th>Total Games played</th>
<th>Mini-Max Wins</th>
<th>Mini-Max Win %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha-Beta</td>
<td>115</td>
<td>53</td>
<td>46%</td>
</tr>
</tbody>
</table>

Table 3: Results of MCTS Algorithm vs UCT Algorithm

<table>
<thead>
<tr>
<th>Algorithm MCTS vs</th>
<th>Total Games played</th>
<th>MCTS Wins</th>
<th>MCTS Win %</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT</td>
<td>85</td>
<td>41</td>
<td>48%</td>
</tr>
</tbody>
</table>

The main evaluation study was to determine how Mini-Max with Alpha-Beta Pruning does against Monte-Carlo Tree Search with UCT for the game of Hasami Shogi. The following are the results for the games played. These results are for UCT with different values for $C = 1, 2$ and $\sqrt{2}$. The resulting win percentages did not differ much for
different C values. MCTS with UCT won 59% of the games played against Alpha-Beta algorithm.

<table>
<thead>
<tr>
<th>Algorithm Mini-Max with Alpha-Beta vs</th>
<th>Total Games played</th>
<th>Mini-Max with Alpha-Beta Wins</th>
<th>Mini-Max with Alpha-Beta Win %</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCTS with UCT</td>
<td>225</td>
<td>93</td>
<td>41%</td>
</tr>
</tbody>
</table>

A limited number of tests were run with values of C = 10 and 100 as listed in Table 5 below. The UCT algorithms winning percentage against Mini-Max with Alpha-Beta went down with Alpha-Beta winning 53% and 57% of the games respectively instead of 41% with values for C = 1, 2 and √2.

<table>
<thead>
<tr>
<th>Algorithm Mini-Max with Alpha-Beta vs</th>
<th>Total Games played</th>
<th>Mini-Max with Alpha-Beta Wins</th>
<th>Mini-Max with Alpha-Beta Win %</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCTS with UCT with C = 10</td>
<td>45</td>
<td>24</td>
<td>53%</td>
</tr>
<tr>
<td>MCTS with UCT with C = 100</td>
<td>35</td>
<td>20</td>
<td>57%</td>
</tr>
</tbody>
</table>

Though the number of simulations done were not enough to conclusively say which algorithm works best for Hasami Shogi, it seems MCTS has a lot of promise. Also a lot can be achieved by further strengthening the evaluation function.
6 CONCLUSIONS AND FUTURE RESEARCH

6.1 Answers to Research Questions

The results of the playouts seem to suggest that MCTS with UCT works best for the game of Hasami Shogi amongst the algorithms implemented. MCTS with UCT won roughly 60% of the games against Alpha-Beta. The contribution of the UCT factor in the end result does not seem to be much but MCTS definitely seems to be the dominant algorithm.

A strong evaluation function is a must for all the algorithms and if more about Hasami Shogi game-intelligence and tricks can be learnt and somehow converted into the evaluation function, it would make really significant difference.

6.2 Conclusions and observations

The main conclusion to be derived from this study is that for such two-player finite position games like Hasami Shogi, Monte Carlo Tree Search implementation seems most promising. UCT implementation did not seem to provide any significant additional benefits over the MCTS algorithm and lower values for the tunable bias parameter C seemed to perform better.

Also in the game of Hasami Shogi, whether the bot was playing first or second did not seem to affect the outcome significantly. Initially the implementation had a time-based move generation. But then some playouts were stopped midway whenever the game-timer expired. Since these were just self-test runs and not against any server based or
other online bots, a limit on the number of playouts instead of timer was implemented for MCTS and UCT algorithms.

The program still is not very strong enough to play intelligent moves all the time and some effort in improving the evaluation function to up the performance is needed.

6.3 Future Research and Development

The algorithms implemented were the basic algorithms with little time spent into optimizing them for game-techniques and intelligent tricks related to the game of Hasami Shogi. Some benefit can also be obtained by having opening and closing strategies which will deviate from the algorithms. Also the evaluation function needs to be developed further and some defensive mechanisms need to be incorporated into the existing code.

Currently the only interface available is the console window which prints the board and moves. Adding a GUI as well as a web interface would be a good enhancement.

The result of the MCTS / UCT algorithm depends heavily on the simulation playouts tried on the child nodes. Some enhancement can be made to increase the number of playouts / simulations tried out on each node making the algorithms more accurate.

Also some research can be made into finding standard protocols so that the bot can be used to run against other Hasami Shogi game-playing bots online.
REFERENCES

1. Hasami Shogi Game Rules [Nov-2011]
   http://www.igfip.com/rules/hasamishogi.html

   http://www.martialtaichi.co.uk/articles/Tafl_Rules.pdf

3. Monte Carlo Tree Search and UCT Algorithm [Nov-2011]
   http://www.mcts.ai/?q=mcts

4. UCT Algorithm Applied to Games with Imperfect Information [Oct-2012]
   http://www.b0n541.net/study/jan_schaefer_uct_algorithm_applied_to_games_with_imperfect_information.pdf

5. Solved Game definitions on Wikipedia [Oct-2012]
   http://en.wikipedia.org/wiki/Solved_game

6. Alpha-Beta Pruning Psuedocode [Nov-2011]
   http://www.wordiq.com/definition/Alpha-beta_pruning#Pseudocode

7. Alpha-Beta pruning examples [Oct-2012]
   http://www.cs.utah.edu/~hal/courses/2009S_AI/Walkthrough/AlphaBeta/

   http://www.stanford.edu/~msirota/soco/minimax.html

   http://ai-depot.com/articles/minimax-explained/

10. A Survey of Monte Carlo Tree Search Methods (March 2012)
    Cameron Browne, Member, IEEE, Edward Powley, Member, IEEE, Daniel Whitehouse, Member, IEEE, Simon Lucas, Senior Member, IEEE, Peter I. Cowling, Member, IEEE, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis and Simon Colton [Oct-2012]

12. Parallel Monte-Carlo Tree Search by Guillaume M.J-B. Chaslot, Mark H.M. Winands, and H. Jaap van den Herik, Games and AI Group, MICC, Faculty of Humanities and Sciences, Universiteit Maastricht, Maastricht, The Netherlands [Nov-2012]
   http://www.personeel.unimaas.nl/m-winands/documents/multithreadedMCTS2.pdf

   http://www.personeel.unimaas.nl/G-chaslot/papers/progressiveMCTS.pdf

   http://www.igfip.com/rules/hasamishogi.html


   http://www.aaai.org/Papers/AIIDE/2008/AIIDE08-036.pdf

17. Evaluation Function on Wikipedia [Nov-2012]

18. Mini-Max Algorithm Game-Tree Image [Nov-2012]
   http://www.cis.temple.edu/~wangp/203-AI/Lecture/Game-1.jpg
// Board Class

using System;
using System.Collections.Generic;
using System.Text;

namespace HSQueenNew
{
    public enum Player { BLACK = -1, WHITE = 1, EMPTY = 0 };

    class Board
    {
        public Player[,] Squares;
        public Player turnToPlay;

        public Board()
        {
            /*
            Squares = new Player[9, 9] {
                { -1, -1, -1, -1, -1, -1, -1, -1, -1 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                { 1, 1, 1, 1, 1, 1, 1, 1, 1 } ];
            */

            Squares = new Player[9, 9] {
                { Player.BLACK, Player.BLACK, Player.BLACK, Player.BLACK, Player.BLACK, Player.BLACK, Player.BLACK, Player.BLACK, Player.BLACK },
                { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY },
                { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY },
                { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY } ,
                { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY } ,
                { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY } ,
                { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY } ,
                { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY } ,
                { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY } ,
            };
        }
    }
}
Squares = new Player[5, 5] { { Player.BLACK, Player.BLACK, Player.BLACK, Player.BLACK, Player.BLACK },
            { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY },
            { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY },
            { Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY, Player.EMPTY },
            { Player.WHITE, Player.WHITE, Player.WHITE, Player.WHITE, Player.WHITE } ];

/*
   turnToPlay = Player.BLACK; // BLACK always plays first
*/

public void PrintToScreen()
{
    for (int x = 0; x < Squares.GetLength(0); x++)
    {
        for (int y = 0; y < Squares.GetLength(1); y++)
        {
            if (Squares[x, y] == Player.BLACK) Console.Write(" B ");
            else if (Squares[x, y] == Player.WHITE) Console.Write(" W ");
            else if (Squares[x, y] == Player.EMPTY) Console.Write(" _ ");
            else Console.Write("ERROR");
        }
        Console.Write("\n");
    }
    Console.Write("\n---------------------------\n");
}
```csharp
public void WriteToFile()
{
    System.IO.File.AppendAllText(@"C:\Users\Bobby\Documents\Visual Studio 2005\Projects\HSQueenNew\OutputText.txt", System.Environment.NewLine);
    for (int x = 0; x < Squares.GetLength(0); x++)
    {
        for (int y = 0; y < Squares.GetLength(1); y++)
        {
            if (Squares[x, y] == Player.BLACK)
            {
                System.IO.File.AppendAllText(@"C:\Users\Bobby\Documents\Visual Studio 2005\Projects\HSQueenNew\OutputText.txt", " B ");
            } else if (Squares[x, y] == Player.WHITE)
            {
                System.IO.File.AppendAllText(@"C:\Users\Bobby\Documents\Visual Studio 2005\Projects\HSQueenNew\OutputText.txt", " W ");
            } else if (Squares[x, y] == Player.EMPTY)
            {
                System.IO.File.AppendAllText(@"C:\Users\Bobby\Documents\Visual Studio 2005\Projects\HSQueenNew\OutputText.txt", " _ ");
            } else
            {
                System.IO.File.AppendAllText(@"C:\Users\Bobby\Documents\Visual Studio 2005\Projects\HSQueenNew\OutputText.txt", " ERROR ");
            }
        }
    } System.IO.File.AppendAllText(@"C:\Users\Bobby\Documents\Visual Studio 2005\Projects\HSQueenNew\OutputText.txt", System.Environment.NewLine);
    System.IO.File.AppendAllText(@"C:\Users\Bobby\Documents\Visual Studio 2005\Projects\HSQueenNew\OutputText.txt", System.Environment.NewLine + "----------
----------------" + System.Environment.NewLine);
}

public void Clone(Board fromB)
{
    // Copies from FromB to this
    Array.Copy(fromB.Squares, Squares, fromB.Squares.Length);
    turnToPlay = fromB.turnToPlay;
}
```
public List<Move> PossibleMoves(Player p)
{
    List<Move> ListMoves = new List<Move>();
    Move tempMove = new Move();

    for (int x = 0; x < Squares.GetLength(0); x++)
    {
        for (int y = 0; y < Squares.GetLength(1); y++)
        {
            if (Squares[x, y] == p)
            {
                tempMove.fromX = x;
                tempMove.fromY = y;

                for (int left = y - 1; left >= 0; left--)
                {
                    if (Squares[x, left] == Player.EMPTY)
                    {
                        tempMove.toX = x;
                        tempMove.toY = left;
                        tempMove.rank = 0;
                        ListMoves.Add(tempMove);
                    }
                    else if (Squares[x, left] == Player.BLACK || Squares[x, left] == Player.WHITE)
                    {
                        break;
                    }
                }

                for (int right = y + 1; right < Squares.GetLength(1); right++)
                {
                    if (Squares[x, right] == Player.EMPTY)
                    {
                        tempMove.toX = x;
                        tempMove.toY = right;
                        tempMove.rank = 0;
                        ListMoves.Add(tempMove);
                    }
                    else if (Squares[x, right] == Player.BLACK || Squares[x, right] == Player.WHITE)
                    {
                        break;
                    }
                }
            }
        }
    }
}

33
for (int up = x - 1; up >= 0; up--)
{
    if (Squares[up, y] == Player.EMPTY)
    {
        tempMove.toX = up;
        tempMove.toY = y;
        tempMove.rank = 0;
        ListMoves.Add(tempMove);
    }
    else if (Squares[up, y] == Player.BLACK || Squares[up, y] == Player.WHITE)
    {
        break;
    }
}

for (int down = x + 1; down < Squares.GetLength(0); down++)
{
    if (Squares[down, y] == Player.EMPTY)
    {
        tempMove.toX = down;
        tempMove.toY = y;
        tempMove.rank = 0;
        ListMoves.Add(tempMove);
    }
    else if (Squares[down, y] == Player.BLACK || Squares[down, y] == Player.WHITE)
    {
        break;
    }
}

return ListMoves;

public bool LegalMovePossible(Player p)
{
    List<Move> PossibleMovesList = new List<Move>();
    PossibleMovesList = PossibleMoves(p);

    if (PossibleMovesList.Count == 0)
    {
        return false;
    }
if (currentMove.toX >= 2)
{  
  if ( //Squares[currentMove.toX, currentMove.toY] == currentP) &&  
    (Squares[currentMove.toX - 1, currentMove.toY] == otherP) &&  
    (Squares[currentMove.toX - 2, currentMove.toY] == currentP))  
  {  
    for (int numcaptured = 1; numcaptured <= 1; numcaptured++)  
    {  
      tempMove.fromX = currentMove.toX - numcaptured;  
      tempMove.fromY = currentMove.toY;  
      tempMove.toX = currentMove.toX - numcaptured;  
      tempMove.toY = currentMove.toY;  
      tempMove.rank = 0;  
      ListCapturedPawns.Add(tempMove);  
    }  
  }  
  else if (currentMove.toX >= 3)  
  {  
    if ( //Squares[currentMove.toX, currentMove.toY] == currentP) &&  
      (Squares[currentMove.toX - 1, currentMove.toY] == otherP) &&  
      (Squares[currentMove.toX - 2, currentMove.toY] == otherP) &&  
      (Squares[currentMove.toX - 3, currentMove.toY] == currentP))  
    {  
      for (int numcaptured = 1; numcaptured <= 2; numcaptured++)  
      {  
        tempMove.fromX = currentMove.toX - numcaptured;  
        tempMove.fromY = currentMove.toY;  
        tempMove.toX = currentMove.toX - numcaptured;  
        tempMove.toY = currentMove.toY;  
        tempMove.rank = 0;  
        ListCapturedPawns.Add(tempMove);  
      }  
    }  
  }  
  else if (currentMove.toX >= 4)  
  {  
    if ( //Squares[currentMove.toX, currentMove.toY] == currentP) &&  
      (Squares[currentMove.toX - 1, currentMove.toY] == otherP) &&  
      (Squares[currentMove.toX - 2, currentMove.toY] == otherP) &&  
      (Squares[currentMove.toX - 3, currentMove.toY] == otherP) &&  
      (Squares[currentMove.toX - 4, currentMove.toY] == currentP))  
    {  
      for (int numcaptured = 1; numcaptured <= 3; numcaptured++)  
      {  
        tempMove.fromX = currentMove.toX - numcaptured;  
        tempMove.fromY = currentMove.toY;  
        tempMove.toX = currentMove.toX - numcaptured;  
        tempMove.toY = currentMove.toY;  
      }  
    }  
  }  
}
tempMove.rank = 0;
ListCapturedPawns.Add(tempMove);
}
}
else if (currentMove.toX >= 5) // This else-if to be commented for 5x5
{
    if ( //Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX - 1, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX - 2, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX - 3, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX - 4, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX - 5, currentMove.toY] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 4; numcaptured++)
        {
            tempMove.fromX = currentMove.toX - numcaptured;
            tempMove.fromY = currentMove.toY;
            tempMove.toX = currentMove.toX - numcaptured;
            tempMove.toY = currentMove.toY;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
else if (currentMove.toX >= 6) // This else-if to be commented for 5x5
{
    if ( //Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX - 1, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX - 2, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX - 3, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX - 4, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX - 5, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX - 6, currentMove.toY] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 5; numcaptured++)
        {
            tempMove.fromX = currentMove.toX - numcaptured;
            tempMove.fromY = currentMove.toY;
            tempMove.toX = currentMove.toX - numcaptured;
            tempMove.toY = currentMove.toY;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
else if (currentMove.toX >= 7) // This else-if to be commented for 5x5
{
if ( //Squares[currentMove.toX, currentMove.toY] == currentP)
    &&
    (Squares[currentMove.toX - 1, currentMove.toY] == otherP)
    &&
    (Squares[currentMove.toX - 2, currentMove.toY] == otherP)
    &&
    (Squares[currentMove.toX - 3, currentMove.toY] == otherP)
    &&
    (Squares[currentMove.toX - 4, currentMove.toY] == otherP)
    &&
    (Squares[currentMove.toX - 5, currentMove.toY] == otherP)
    &&
    (Squares[currentMove.toX - 6, currentMove.toY] == otherP)
    &&
    (Squares[currentMove.toX - 7, currentMove.toY] == otherP)
    {
        for (int numcaptured = 1; numcaptured <= 6; numcaptured++)
        {
            tempMove.fromX = currentMove.toX - numcaptured;
            tempMove.fromY = currentMove.toY;
            tempMove.toX = currentMove.toX - numcaptured;
            tempMove.toY = currentMove.toY;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
else if (currentMove.toX >= 8) // This else-if to be commented for 5x5
    {
        if ( //Squares[currentMove.toX, currentMove.toY] == currentP)
            &&
            (Squares[currentMove.toX - 1, currentMove.toY] == otherP)
            &&
            (Squares[currentMove.toX - 2, currentMove.toY] == otherP)
            &&
            (Squares[currentMove.toX - 3, currentMove.toY] == otherP)
            &&
            (Squares[currentMove.toX - 4, currentMove.toY] == otherP)
            &&
            (Squares[currentMove.toX - 5, currentMove.toY] == otherP)
            &&
            (Squares[currentMove.toX - 6, currentMove.toY] == otherP)
            &&
            (Squares[currentMove.toX - 7, currentMove.toY] == otherP)
(Squares[currentMove.toX - 8, currentMove.toY] ==
currentP))
{
    for (int numcaptured = 1; numcaptured <= 7; numcaptured++)
    {
        tempMove.fromX = currentMove.toX - numcaptured;
        tempMove.fromY = currentMove.toY;
        tempMove.toX = currentMove.toX - numcaptured;
        tempMove.toY = currentMove.toY;
        tempMove.rank = 0;
        ListCapturedPawns.Add(tempMove);
    }
}
} // end of check the left side

// check the right side
if (currentMove.toX <= 6 /*2 - for 5x5*/) {
    if (//(Squares[currentMove.toX, currentMove.toY] == currentP) &&
       (Squares[currentMove.toX + 1, currentMove.toY] == otherP) &&
       (Squares[currentMove.toX + 2, currentMove.toY] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 1; numcaptured++)
        {
            tempMove.fromX = currentMove.toX + numcaptured;
            tempMove.fromY = currentMove.toY;
            tempMove.toX = currentMove.toX + numcaptured;
            tempMove.toY = currentMove.toY;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
    else if (currentMove.toX <= 5 /*1 - for 5x5*/) {
        if (//(Squares[currentMove.toX, currentMove.toY] == currentP) &&
            (Squares[currentMove.toX + 1, currentMove.toY] == otherP) &&
            (Squares[currentMove.toX + 2, currentMove.toY] == otherP) &&
            (Squares[currentMove.toX + 3, currentMove.toY] == currentP))
        {
            // code
for (int numcaptured = 1; numcaptured <= 2; numcaptured++)
{
    tempMove.fromX = currentMove.toX + numcaptured;
    tempMove.fromY = currentMove.toY;
    tempMove.toX = currentMove.toX + numcaptured;
    tempMove.toY = currentMove.toY;
    tempMove.rank = 0;
    ListCapturedPawns.Add(tempMove);
}
}
else if (currentMove.toX <= 4 /*0 - for 5x5*/)
{
    if ((Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX + 1, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 2, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 3, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 4, currentMove.toY] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 3; numcaptured++)
        {
            tempMove.fromX = currentMove.toX + numcaptured;
            tempMove.fromY = currentMove.toY;
            tempMove.toX = currentMove.toX + numcaptured;
            tempMove.toY = currentMove.toY;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
}
else if (currentMove.toX <= 3) // This else-if to be commented for 5x5
{
    if ((Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX + 1, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 2, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 3, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 4, currentMove.toY] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 4; numcaptured++)
        {
            tempMove.fromX = currentMove.toX + numcaptured;
            tempMove.fromY = currentMove.toY;
            tempMove.toX = currentMove.toX + numcaptured;
            tempMove.toY = currentMove.toY;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
}
}  
else if (currentMove.toX <= 2) // This else-if to be commented for 5x5  
{  
    if (//(Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX + 1, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 2, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 3, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 4, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 5, currentMove.toY] == otherP) &&
        (Squares[currentMove.toX + 6, currentMove.toY] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 5; numcaptured++)
        {
            tempMove.fromX = currentMove.toX + numcaptured;
            tempMove.fromY = currentMove.toY;
            tempMove.toX = currentMove.toX + numcaptured;
            tempMove.toY = currentMove.toY;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
}
else if (currentMove.toX <= 1) // This else-if to be commented for 5x5
{  
    if (//(Squares[currentMove.toX, currentMove.toY] == currentP)
        &&
        (Squares[currentMove.toX + 1, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 2, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 3, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 4, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 5, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 6, currentMove.toY] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 6; numcaptured++)
        {
            tempMove.fromX = currentMove.toX + numcaptured;
            tempMove.fromY = currentMove.toY;
            tempMove.toX = currentMove.toX + numcaptured;
            tempMove.toY = currentMove.toY;
        }
    }
}
tempMove.rank = 0;
ListCapturedPawns.Add(tempMove);
}
}
else if (currentMove.toX <= 0) // This else-if to be commented for 5x5
{
    if (//(Squares[currentMove.toX, currentMove.toY] == currentP)
        &&
        (Squares[currentMove.toX + 1, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 2, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 3, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 4, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 5, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 6, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 7, currentMove.toY] == otherP)
        &&
        (Squares[currentMove.toX + 8, currentMove.toY] ==
currentP))
    {
        for (int numcaptured = 1; numcaptured <= 7; numcaptured++)
        {
            tempMove.fromX = currentMove.toX + numcaptured;
tempMove.fromY = currentMove.toY;
tempMove.toX = currentMove.toX + numcaptured;
tempMove.toY = currentMove.toY;
tempMove.rank = 0;
ListCapturedPawns.Add(tempMove);
        }
    }
}
else if (currentMove.toX == 0) // This else-if to be commented for 5x5
{
    if (//(Squares[currentMove.toX, currentMove.toY] == currentP) ||
        (Squares[currentMove.toX + 1, currentMove.toY] ==
otherP) ||
        (Squares[currentMove.toX + 2, currentMove.toY] == otherP) ||
        (Squares[currentMove.toX + 3, currentMove.toY] == otherP) ||
        (Squares[currentMove.toX + 4, currentMove.toY] == otherP) ||
        (Squares[currentMove.toX + 5, currentMove.toY] == otherP) ||
        (Squares[currentMove.toX + 6, currentMove.toY] == otherP) ||
        (Squares[currentMove.toX + 7, currentMove.toY] == otherP) ||
        (Squares[currentMove.toX + 8, currentMove.toY] ==
currentP))
    {
        for (int numcaptured = 1; numcaptured <= 7; numcaptured++)
        {
            tempMove.fromX = currentMove.toX + numcaptured;
tempMove.fromY = currentMove.toY;
tempMove.toX = currentMove.toX + numcaptured;
tempMove.toY = currentMove.toY;
tempMove.rank = 0;
ListCapturedPawns.Add(tempMove);
        }
    }
}
}
} // end of check the right side

// check the up side
if (currentMove.toY >= 2)
if (Squares[currentMove.toX, currentMove.toY] == currentP) &&
(Squares[currentMove.toX, currentMove.toY - 1] == otherP) &&
(Squares[currentMove.toX, currentMove.toY - 2] == currentP))
{
  for (int numcaptured = 1; numcaptured <= 1; numcaptured++)
  {
    tempMove.fromX = currentMove.toX;
    tempMove.fromY = currentMove.toY - numcaptured;
    tempMove.toX = currentMove.toX;
    tempMove.toY = currentMove.toY - numcaptured;
    tempMove.rank = 0;
    ListCapturedPawns.Add(tempMove);
  }
}
else if (currentMove.toY >= 3)
{
  if (Squares[currentMove.toX, currentMove.toY] == currentP) &&
(Squares[currentMove.toX, currentMove.toY - 1] == otherP) &&
(Squares[currentMove.toX, currentMove.toY - 2] == otherP) &&
(Squares[currentMove.toX, currentMove.toY - 3] == currentP))
  {
    for (int numcaptured = 1; numcaptured <= 2; numcaptured++)
    {
      tempMove.fromX = currentMove.toX;
      tempMove.fromY = currentMove.toY - numcaptured;
      tempMove.toX = currentMove.toX;
      tempMove.toY = currentMove.toY - numcaptured;
      tempMove.rank = 0;
      ListCapturedPawns.Add(tempMove);
    }
  }
else if (currentMove.toY >= 4)
{
  if (Squares[currentMove.toX, currentMove.toY] == currentP) &&
(Squares[currentMove.toX, currentMove.toY - 1] == otherP) &&
(Squares[currentMove.toX, currentMove.toY - 2] == otherP) &&
(Squares[currentMove.toX, currentMove.toY - 3] == otherP) &&
(Squares[currentMove.toX, currentMove.toY - 4] == currentP))
  {
    for (int numcaptured = 1; numcaptured <= 3; numcaptured++)
    {
      tempMove.fromX = currentMove.toX;
      tempMove.fromY = currentMove.toY - numcaptured;
      tempMove.toX = currentMove.toX;
      tempMove.toY = currentMove.toY - numcaptured;
    }
  }
tempMove.rank = 0;
ListCapturedPawns.Add(tempMove);
}
}
else if (currentMove.toY >= 5) // This else-if to be commented for 5x5
{
    if (//(Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX, currentMove.toY - 1] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY - 2] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY - 3] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY - 4] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY - 5] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 4; numcaptured++)
        {
            tempMove.fromX = currentMove.toX;
            tempMove.fromY = currentMove.toY - numcaptured;
            tempMove.toX = currentMove.toX;
            tempMove.toY = currentMove.toY - numcaptured;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
else if (currentMove.toY >= 6) // This else-if to be commented for 5x5
{
    if (//(Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX, currentMove.toY - 1] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY - 2] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY - 3] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY - 4] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY - 5] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 5; numcaptured++)
        {
            tempMove.fromX = currentMove.toX;
            tempMove.fromY = currentMove.toY - numcaptured;
            tempMove.toX = currentMove.toX;
            tempMove.toY = currentMove.toY - numcaptured;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
}
if ( // (Squares[currentMove.toX, currentMove.toY] == currentP)
&&
    (Squares[currentMove.toX, currentMove.toY - 1] == otherP)
&&
    (Squares[currentMove.toX, currentMove.toY - 2] == otherP)
&&
    (Squares[currentMove.toX, currentMove.toY - 3] == otherP)
&&
    (Squares[currentMove.toX, currentMove.toY - 4] == otherP)
&&
    (Squares[currentMove.toX, currentMove.toY - 5] == otherP)
&&
    (Squares[currentMove.toX, currentMove.toY - 6] == otherP)
&&
    (Squares[currentMove.toX, currentMove.toY - 7] == currentP))
{
    for (int numcaptured = 1; numcaptured <= 6; numcaptured++)
    {
        tempMove.fromX = currentMove.toX;
        tempMove.fromY = currentMove.toY - numcaptured;
        tempMove.toX = currentMove.toX;
        tempMove.toY = currentMove.toY - numcaptured;
        tempMove.rank = 0;
        ListCapturedPawns.Add(tempMove);
    }
}
else if (currentMove.toY >= 8) // This else-if to be commented for 5x5
{
    if ( // (Squares[currentMove.toX, currentMove.toY] == currentP)
&&
        (Squares[currentMove.toX, currentMove.toY - 1] == otherP)
&&
        (Squares[currentMove.toX, currentMove.toY - 2] == otherP)
&&
        (Squares[currentMove.toX, currentMove.toY - 3] == otherP)
&&
        (Squares[currentMove.toX, currentMove.toY - 4] == otherP)
&&
        (Squares[currentMove.toX, currentMove.toY - 5] == otherP)
&&
        (Squares[currentMove.toX, currentMove.toY - 6] == otherP)
&&
        (Squares[currentMove.toX, currentMove.toY - 7] == otherP)
(Squares[currentMove.toX, currentMove.toY - 8] == currentP))
{
    for (int numcaptured = 1; numcaptured <= 7; numcaptured++)
    {
        tempMove.fromX = currentMove.toX;
        tempMove.fromY = currentMove.toY - numcaptured;
        tempMove.toX = currentMove.toX;
        tempMove.toY = currentMove.toY - numcaptured;
        tempMove.rank = 0;
        ListCapturedPawns.Add(tempMove);
    }
}
} // end of check the up side

// check the down side
if (currentMove.toY <= 6 /*2 - for 5x5*/)
{
    if (//(Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX, currentMove.toY + 1] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 2] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 1; numcaptured++)
        {
            tempMove.fromX = currentMove.toX;
            tempMove.fromY = currentMove.toY + numcaptured;
            tempMove.toX = currentMove.toX;
            tempMove.toY = currentMove.toY + numcaptured;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
} else if (currentMove.toY <= 5 /*1 - for 5x5*/)
{
    if (//(Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX, currentMove.toY + 1] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 2] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 3] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 2; numcaptured++)
        {
{ 
    tempMove.fromX = currentMove.toX;
tempMove.fromY = currentMove.toY + numcaptured;
tempMove.toX = currentMove.toX;
tempMove.toY = currentMove.toY + numcaptured;
tempMove.rank = 0;
ListCapturedPawns.Add(tempMove);
}

else if (currentMove.toY <= 4 /*0 - for 5x5*/) 
{
    if ( //((Squares[currentMove.toX, currentMove.toY] == currentP) &
        (Squares[currentMove.toX, currentMove.toY + 1] == otherP) &
        (Squares[currentMove.toX, currentMove.toY + 2] == otherP) &
        (Squares[currentMove.toX, currentMove.toY + 3] == otherP) &
        (Squares[currentMove.toX, currentMove.toY + 4] == currentP))
    
    for (int numcaptured = 1; numcaptured <= 3; numcaptured++)
    { 
        tempMove.fromX = currentMove.toX;
tempMove.fromY = currentMove.toY + numcaptured;
tempMove.toX = currentMove.toX;
tempMove.toY = currentMove.toY + numcaptured;
tempMove.rank = 0;
ListCapturedPawns.Add(tempMove);
    }
}

else if (currentMove.toY <= 3) // This else-if to be commented for 5x5
{
    if ( //((Squares[currentMove.toX, currentMove.toY] == currentP) &
        (Squares[currentMove.toX, currentMove.toY + 1] == otherP) &
        (Squares[currentMove.toX, currentMove.toY + 2] == otherP) &
        (Squares[currentMove.toX, currentMove.toY + 3] == otherP) &
        (Squares[currentMove.toX, currentMove.toY + 4] == currentP))
    
    for (int numcaptured = 1; numcaptured <= 4; numcaptured++)
    { 
        tempMove.fromX = currentMove.toX;
tempMove.fromY = currentMove.toY + numcaptured;
tempMove.toX = currentMove.toX;
tempMove.toY = currentMove.toY + numcaptured;
tempMove.rank = 0;
ListCapturedPawns.Add(tempMove);
    }
}
else if (currentMove.toY <= 2) // This else-if to be commented for 5x5
{
    if ( // (Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX, currentMove.toY + 1] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 2] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 3] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 4] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 5] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 6] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 5; numcaptured++)
        {
            tempMove.fromX = currentMove.toX;
            tempMove.fromY = currentMove.toY + numcaptured;
            tempMove.toX = currentMove.toX;
            tempMove.toY = currentMove.toY + numcaptured;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
}
else if (currentMove.toY <= 1) // This else-if to be commented for 5x5
{
    if ( // (Squares[currentMove.toX, currentMove.toY] == currentP) &&
        (Squares[currentMove.toX, currentMove.toY + 1] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 2] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 3] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 4] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 5] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 6] == otherP) &&
        (Squares[currentMove.toX, currentMove.toY + 7] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 6; numcaptured++)
        {
            tempMove.fromX = currentMove.toX;
            tempMove.fromY = currentMove.toY + numcaptured;
            tempMove.toX = currentMove.toX;
            tempMove.toY = currentMove.toY + numcaptured;
            tempMove.rank = 0;
        }
    }
}
ListCapturedPawns.Add(tempMove);

else if (currentMove.toY <= 0) // This else-if to be commented for 5x5
{
    if (Squares[currentMove.toX, currentMove.toY] == currentP)
        && (Squares[currentMove.toX, currentMove.toY + 1] == otherP)
        && (Squares[currentMove.toX, currentMove.toY + 2] == otherP)
        && (Squares[currentMove.toX, currentMove.toY + 3] == otherP)
        && (Squares[currentMove.toX, currentMove.toY + 4] == otherP)
        && (Squares[currentMove.toX, currentMove.toY + 5] == otherP)
        && (Squares[currentMove.toX, currentMove.toY + 6] == otherP)
        && (Squares[currentMove.toX, currentMove.toY + 7] == otherP)
        && (Squares[currentMove.toX, currentMove.toY + 8] == currentP))
    {
        for (int numcaptured = 1; numcaptured <= 7; numcaptured++)
        {
            tempMove.fromX = currentMove.toX;
            tempMove.fromY = currentMove.toY + numcaptured;
            tempMove.toX = currentMove.toX;
            tempMove.toY = currentMove.toY + numcaptured;
            tempMove.rank = 0;
            ListCapturedPawns.Add(tempMove);
        }
    }
}

// end of check the down side

return ListCapturedPawns;
public Player DidAnyoneWin()
{
    int countWhite = 0;
    int countBlack = 0;

    for (int x = 0; x < Squares.GetLength(0); x++)
    {
        for (int y = 0; y < Squares.GetLength(1); y++)
        {
            if (Squares[x, y] == Player.BLACK) countBlack++;
            else if (Squares[x, y] == Player.WHITE) countWhite++;
        }
    }

    if (countWhite < 2 || LegalMovePossible(Player.WHITE) == false) return Player.BLACK;
    else if (countBlack < 2 || LegalMovePossible(Player.BLACK) == false) return Player.WHITE;
    else return Player.EMPTY;
}
}
// AI Class

using System;
using System.Collections.Generic;
using System.Text;

namespace HSQueenNew
{
    public struct Move
    {
        public int fromX;
        public int fromY;
        public int toX;
        public int toY;
        public int rank; //MCTS/UCT and MiniMax/AlphaBeta use this differently
    }
}

class AI
{
    public static int depthcount = 0;

    public Move RandomMove(Board b, Player p)
    {
        Move bestMove = new Move();

        List<Move> ListPossibleMoves = new List<Move>();
        ListPossibleMoves = b.PossibleMoves(p);
        // Evaluation Function - identify more promising moves amongst them
        ListPossibleMoves = MoveEvaluation(b, p, ListPossibleMoves);

        int bestRank = 0;
        for (int q = 0; q < ListPossibleMoves.Count; q++)
        {
            if (ListPossibleMoves[q].rank > bestRank)
            {
                bestRank = ListPossibleMoves[q].rank;
            }
        }

        while (true)
        {
            Random r = new Random();
            bestMove = ListPossibleMoves[r.Next(0, ListPossibleMoves.Count)];
            if (bestMove.rank >= 0.7 * bestRank)
            {
            }
        }
    }
}
return bestMove;
}
}

public Move RandomMoveMCTS(Board b, Player p, Tree t)
{
    Move bestMove = new Move();

    List<Move> ListPossibleMoves = new List<Move>();
    ListPossibleMoves = b.PossibleMoves(p);
    // Evaluation Function - identify more promising moves amongst them
    ListPossibleMoves = MoveEvaluation(b, p, ListPossibleMoves);

    // Add all the legal possible moves to the tree where the node pointed to by the
gamePtr is the parent
    for (int m = 0; m < ListPossibleMoves.Count; m++)
    {
        t.AddChildNode(t.gamePtr, ListPossibleMoves[m], p);
    }

    int bestRank = 0;
    for (int q = 0; q < ListPossibleMoves.Count; q++)
    {
        if (ListPossibleMoves[q].rank > bestRank)
        {
            bestRank = ListPossibleMoves[q].rank;
        }
    }

    while (true)
    {
        Random r = new Random();
        bestMove = ListPossibleMoves[r.Next(0, ListPossibleMoves.Count)];
        if (bestMove.rank >= 0.7 * bestRank)
        {
            return bestMove;
        }
    }
}

public Move MiniMaxMove(Board b, Player p)
{
    int numCaptured = 0;
    Move? bestMove = null; //parameter T
List<Move> listPossibleMoves = new List<Move>();
listPossibleMoves = b.PossibleMoves(p);

Board copyOfb = new Board();

for (int i = 0; i < listPossibleMoves.Count; i++)
{
    copyOfb.Clone(b);
    Move newMove = listPossibleMoves[i];
    int numCapturedInThisMove = 0;

    // Use the return value of numCapturedPawns for assigning the rank for the move
    numCapturedInThisMove = copyOfb.PlayThisMove(newMove);

    // Nobody has won and there are more possible moves left to evaluate
    if (copyOfb.DidAnyoneWin() == Player.EMPTY &&
        (listPossibleMoves.Count - 1) > 0 &&
        depthcount < 25)
    {
        Player invertP = new Player();
        if (p == Player.WHITE) invertP = Player.BLACK;
        else if (p == Player.BLACK) invertP = Player.WHITE;

        depthcount++;
        //Console.Write("n" + depthcount);
        Move tempMove = MiniMaxMove(copyOfb, invertP);
        newMove.rank = tempMove.rank;
    }
    else
    {
        if (copyOfb.DidAnyoneWin() == Player.BLACK)
            newMove.rank = 10; // Max rank for winning move
        else if (copyOfb.DidAnyoneWin() == Player.WHITE)
            newMove.rank = -10; //Max rank for winning move
        else if (copyOfb.DidAnyoneWin() == Player.EMPTY)
        {
            newMove.rank = numCapturedInThisMove;
            if (p == Player.WHITE) newMove.rank = -newMove.rank;
        }
    }

    // If new move is better than previous move, take it
    if (bestMove == null ||
        (p == Player.BLACK && newMove.rank > ((Move)bestMove).rank) ||
        (p == Player.WHITE && newMove.rank < ((Move)bestMove).rank))
public Move AlphaBetaPruningMove(Board b, Player p, int alpha, int beta)
{
    int numCaptured = 0;
    Move? bestMove = null; //parameter T
    List<Move> listPossibleMoves = new List<Move>();
    listPossibleMoves = b.PossibleMoves(p);
    // Evaluation Function - to somehow identify more promising moves amongst them
    listPossibleMoves = MoveEvaluation(b, p, listPossibleMoves);
    Board copyOfb = new Board();
    for (int i = 0; i < listPossibleMoves.Count; i++)
    {
        copyOfb.Clone(b);
        Move newMove = listPossibleMoves[i];
        int numCapturedInThisMove = 0;
        // Use the return value of numCapturedPawns for assigning the rank for the move
        numCapturedInThisMove = copyOfb.PlayThisMove(newMove);
        // Nobody has won and there are more possible moves left to evaluate
        if (copyOfb.DidAnyoneWin() == Player.EMPTY &&
            (listPossibleMoves.Count - 1) > 0 &&
            depthcount < 25)
        {
            Player invertP = new Player();
            if (p == Player.WHITE) invertP = Player.BLACK;
            else if (p == Player.BLACK) invertP = Player.WHITE;
            depthcount++;
            //Console.Write("\n" + depthcount);
            Move tempMove = AlphaBetaPruningMove(copyOfb, invertP, alpha, beta);
            newMove.rank = tempMove.rank;
        }
    }
    return bestMove;
}
else
{
    if (copyOfb.DidAnyoneWin() == Player.BLACK)
        newMove.rank = 10; // Max rank for winning move
    else if (copyOfb.DidAnyoneWin() == Player.WHITE)
        newMove.rank = -10; // Max rank for winning move
    else if (copyOfb.DidAnyoneWin() == Player.EMPTY)
    {
        newMove.rank = numCapturedInThisMove;
        if (p == Player.WHITE) newMove.rank = -newMove.rank;
    }
}

if (p == Player.BLACK)
{
    if (alpha < newMove.rank)
    {
        alpha = newMove.rank;
        bestMove = newMove;
        if (alpha >= beta) {break;}
    }
}
else if (p == Player.WHITE)
{
    if (beta > -newMove.rank)
    {
        beta = -newMove.rank;
        bestMove = newMove;
        if (alpha >= beta) {break;}
    }
}

// If new move is better than previous move, take it
if (bestMove == null ||
    (p == Player.BLACK && newMove.rank > ((Move)bestMove).rank) ||
    (p == Player.WHITE && newMove.rank < ((Move)bestMove).rank))
{
    bestMove = newMove;
}
}

depthcount = 0;
return (Move)bestMove;

}
Node bestChild = null;

// EXPANSION PHASE - Add all the possible legal moves from this position to the tree
List<Move> listPossibleMoves = new List<Move>();
listPossibleMoves = b.PossibleMoves(p);
// Evaluation Function - Identify more promising moves amongst them
listPossibleMoves = MoveEvaluation(b, p, listPossibleMoves);

for (int m = 0; m < listPossibleMoves.Count; m++)
{
    t.AddChildNode(t.gamePtr, listPossibleMoves[m], p);
}

for (int numGamesPerMove = 0; numGamesPerMove < MAXGAMESPERMOVE; numGamesPerMove++)
{
    // SELECTION PHASE - Pick one of the unexplored nodes / exploit existing nodes
    Node[] currentNodes = t.GetAllChildNodes(t.gamePtr);

    int bestRank = 0;
    for (int q = 0; q < currentNodes.Length; q++)
    {
        if (currentNodes[q].move.rank > bestRank)
        {
            bestRank = currentNodes[q].move.rank;
        }
    }

    int i = 0;
    while (true)
    {
        Random r = new Random();
        i = r.Next(0, currentNodes.Length);
        if (currentNodes[i].move.rank >= 0.7 * bestRank)
        {
            break;
        }
    }

    // SIMULATION PHASE
    // For the move picked, simulate a random playout on copyOfB board
    Board copyOfb = new Board();
    copyOfb.Clone(b);
copyOfb.PlayThisMove(currentNodes[i].move);
//copyOfb.PrintToScreen();

// Assign the simulationPtr to the selected move
  t.simulationPtr = currentNodes[i];

while (copyOfb.LegalMovePossible(copyOfb.turnToPlay) &&
        copyOfb.DidAnyoneWin() == Player.EMPTY)
{
    Move tempMove = RandomMove(copyOfb, copyOfb.turnToPlay);
    copyOfb.PlayThisMove(tempMove);
    //copyOfb.PrintToScreen();

    //Add the simulated move to the tree
    t.AddChildNode(t.simulationPtr, tempMove, copyOfb.turnToPlay);

    // Move the simulatedPtr further down
    Node[] simulatedNodes = t.GetAllChildNodes(t.simulationPtr);
    for (int n = 0; n < simulatedNodes.Length; n++)
    {
        if (simulatedNodes[n].move.fromX == tempMove.fromX &&
            simulatedNodes[n].move.fromY == tempMove.fromY &&
            simulatedNodes[n].move.toX == tempMove.toX &&
            simulatedNodes[n].move.toY == tempMove.toY)
        {
            t.simulationPtr = simulatedNodes[n];
        }
    }
}

while (copyOfb.DidAnyoneWin() != Player.EMPTY &&
        t.simulationPtr != null)
{
    if (t.simulationPtr.isNodeBlackWhite == copyOfb.DidAnyoneWin())
    {
        // BACKPROPAGATION PHASE
        t.simulationPtr.numWins++;
    }

    // BACKPROPAGATION PHASE
    t.simulationPtr.numVisits++;
    t.simulationPtr = t.simulationPtr.parent;
}

if (bestChild == null || bestChild.numVisits == 0 ||
    currentNodes[i].numWins / currentNodes[i].numVisits >
    bestChild.numWins / bestChild.numVisits)
{

bestChild = currentNodes[i];
if (bestChild.numVisits != 0) {
    Console.WriteLine("Wins/Visits Value of best child - " +
    bestChild.numWins / bestChild.numVisits);
}
}

return bestChild.move;

public Move UpperConfidenceBoundsMove(Board b, Player p, Tree t) {
    int MAXGAMESPERMOVE = 75;

    Node bestChild = null;

    // EXPANSION PHASE - Add all the possible legal moves from this position to the tree
    List<Move> listPossibleMoves = new List<Move>();
    listPossibleMoves = b.PossibleMoves(p);

    // Evaluation Function - Identify more promising moves amongst them
    listPossibleMoves = MoveEvaluation(b, p, listPossibleMoves);

    for (int m = 0; m < listPossibleMoves.Count; m++)
    {
        t.AddChildNode(t.gamePtr, listPossibleMoves[m], p);
    }

    for (int numGamesPerMove = 0; numGamesPerMove < MAXGAMESPERMOVE; numGamesPerMove++)
    {
        // SELECTION PHASE - Pick one of the unexplored nodes / exploit existing nodes
        Node[] currentNodes = t.GetAllChildNodes(t.gamePtr);

        int bestRank = 0;
        for (int q = 0; q < currentNodes.Length; q++)
        {
            if (currentNodes[q].move.rank > bestRank)
            {
                bestRank = currentNodes[q].move.rank;
            }
        }
    }
}
int i = 0;
while (true)
{
    Random r = new Random();
    i = r.Next(0, currentNodes.Length);
    if (currentNodes[i].move.rank >= 0.7 * bestRank)
    {
        break;
    }
}

// SIMULATION PHASE
// For the move picked, simulate a random playout on copyOfB board
Board copyOfb = new Board();
copyOfb.Clone(b);
copyOfb.PlayThisMove(currentNodes[i].move);
//copyOfb.PrintToScreen();

// Assign the simulationPtr to the selected move
t.simulationPtr = currentNodes[i];

while (copyOfb.LegalMovePossible(copyOfb.turnToPlay) &&
    copyOfb.DidAnyoneWin() == Player.EMPTY)
{
    Move tempMove = RandomMove(copyOfb, copyOfb.turnToPlay);
    copyOfb.PlayThisMove(tempMove);
    //copyOfb.PrintToScreen();

    //Add the simulated move to the tree
t.AddChildNode(t.simulationPtr, tempMove, copyOfb.turnToPlay);

    // Move the simulatedPtr further down
Node[] simulatedNodes = t.GetAllChildNodes(t.simulationPtr);
for (int n = 0; n < simulatedNodes.Length; n++)
{
    if (simulatedNodes[n].move.fromX == tempMove.fromX &&
        simulatedNodes[n].move.fromY == tempMove.fromY &&
        simulatedNodes[n].move.toX == tempMove.toX &&
        simulatedNodes[n].move.toY == tempMove.toY)
    {
        t.simulationPtr = simulatedNodes[n];
    }
}
}
while (copyOfb.DidAnyoneWin() != Player.EMPTY &&
t.simulationPtr != null
{
  if (t.simulationPtr.isNodeBlackWhite == copyOfb.DidAnyoneWin())
  {
    // BACKPROPAGATION PHASE
    t.simulationPtr.numWins++;
  }
  // BACKPROPAGATION PHASE
  t.simulationPtr.numVisits++;
  t.simulationPtr = t.simulationPtr.parent;
}

// UCT value
if (bestChild == null || bestChild.numVisits == 0)
{
  bestChild = currentNodes[i];
}
else
{
  double C = 2;
  double bestChildUCTvalue = 0;
  double currentNodeUCTvalue = 0;
  if (currentNodeUCTvalue >= bestChildUCTvalue)
  {
    bestChild = currentNodes[i];
    Console.Write("Number of VISITS of best child = " + bestChild.numVisits);
    Console.Write("\n");
    Console.Write("Number of WINS of best child = " + bestChild.numWins);
    Console.Write("\n");
    Console.Write("UCT Value of best child with C = " + C + "is " + bestChildUCTvalue);
    Console.Write("\n");
  }
}
return bestChild.move;
}

public List<Move> MoveEvaluation(Board b, Player p, List<Move> listPossibleMoves)
{
    Move[] listPossibleMovesRanked = listPossibleMoves.ToArray();
    Player opponentP = new Player();
    if (p == Player.WHITE) opponentP = Player.BLACK;
    else if (p == Player.BLACK) opponentP = Player.WHITE;

    // Check if PossibleMove will put a piece next to opponent's piece
    for (int i = 0; i < listPossibleMovesRanked.Length; i++)
    {
        // Check if square LEFT of [toX,toY] occupied by opponent
        if (listPossibleMovesRanked[i].toY > 0 &&
            b.Squares[listPossibleMovesRanked[i].toX, listPossibleMovesRanked[i].toY - 1] == opponentP)
            listPossibleMovesRanked[i].rank += 10;
        // Check if square LEFT-LEFT of [toX,toY] occupied by self - This will lead to capture of 1 piece
        if (listPossibleMovesRanked[i].toY > 1 &&
            b.Squares[listPossibleMovesRanked[i].toX, listPossibleMovesRanked[i].toY - 2] == p)
            listPossibleMovesRanked[i].rank += 30;
    }
    // Check if square TOP of [toX,toY] occupied by opponent
    if (listPossibleMovesRanked[i].toX > 0 &&
        b.Squares[listPossibleMovesRanked[i].toX, listPossibleMovesRanked[i].toY + 1] == opponentP)
        listPossibleMovesRanked[i].rank += 10;
    // Check if square TOP-LEFT of [toX,toY] occupied by self - This will lead to capture of 1 piece
    if (listPossibleMovesRanked[i].toX < 7 &&
        b.Squares[listPossibleMovesRanked[i].toX, listPossibleMovesRanked[i].toY + 2] == opponentP)
        listPossibleMovesRanked[i].rank += 30;
    // Check if square TOP of [toX,toY] occupied by opponent
    if (listPossibleMovesRanked[i].toX > 0 &&
b.Squares[listPossibleMovesRanked[i].toX - 1, listPossibleMovesRanked[i].toY] == opponentP)
{
    listPossibleMovesRanked[i].rank += 10;
    // Check if square TOP-TOP of [toX, toY] occupied by self - This will lead to capture of 1 piece
    if (listPossibleMovesRanked[i].toX > 1 &&
        b.Squares[listPossibleMovesRanked[i].toX - 2, listPossibleMovesRanked[i].toY] == opponentP)
    {
        listPossibleMovesRanked[i].rank += 30;
    }
}
// Check if square BOTTOM of [toX, toY] occupied by opponent
if (listPossibleMovesRanked[i].toX < 8 &&
    b.Squares[listPossibleMovesRanked[i].toX + 1, listPossibleMovesRanked[i].toY] == opponentP)
{
    listPossibleMovesRanked[i].rank += 10;
    // Check if square BOTTOM-BOTTOM of [toX, toY] occupied by self - This will lead to capture of 1 piece
    if (listPossibleMovesRanked[i].toX < 7 &&
        b.Squares[listPossibleMovesRanked[i].toX + 2, listPossibleMovesRanked[i].toY] == opponentP)
    {
        listPossibleMovesRanked[i].rank += 30;
    }
}
return new List<Move>(listPossibleMovesRanked);
// Tree Class and Node Class

using System;
using System.Collections.Generic;
using System.Text;

namespace HSQueenNew
{
    public class Node
    {
        public Node parent;
        public Node[] child;
        public Move move;
        public int numWins;
        public int numVisits;
        public bool wasThisPlayed;
        public Player isNodeBlackWhite;
        // Used for creating the ROOT node of the whole tree
        public Node()
        {
            Move? move = null; //parameter T
            parent = null;
            numWins = -1;
            numVisits = -1;
            wasThisPlayed = true; // root is always played
            isNodeBlackWhite = Player.EMPTY; // root of tree neither BLACK nor WHITE
        }

        public Node(Move m, Node p, Player player)
        {
            move = m;
            parent = p;
            numWins = 0;
            numVisits = 0;
            wasThisPlayed = false;
            isNodeBlackWhite = player;
        }
    }

    public class Tree
    {
        public Node root;
        public Node gamePtr;
        public Node simulationPtr;
        public Tree()
        {
        }
    }
}
root = new Node();
gamePtr = root;
simulationPtr = root;

public void AddChildNode(Node parent, Move m, Player player)
{
    bool moveExists = false;
    for (int i = 0; parent.child != null && i < parent.child.Length; i++)
    {
        if (parent.child[i].move.fromX == m.fromX &&
            parent.child[i].move.fromY == m.fromY &&
            parent.child[i].move.toX == m.toX &&
            parent.child[i].move.toY == m.toY)
        {
            moveExists = true;
            break;
        }
    }
    if (!moveExists)
    {
        Node[] tempParent;
        if (parent.child != null)
        {
            tempParent = new Node[parent.child.Length + 1];
            for (int k = 0; k < parent.child.Length; k++)
            {
                tempParent[k] = parent.child[k];
            }
        }
        else
        {
            tempParent = new Node[1];
        }
        tempParent[tempParent.Length - 1] = new Node(m, parent, player);
        parent.child = tempParent;
    }
    return;
}

public Node[] GetAllChildNodes(Node parent)
{
    return parent.child;
}
using System;
using System.Collections.Generic;
using System.Text;

namespace HSQueenNew
{
    class Program
    {
        static void Main(string[] args)
        {
            Board b = new Board();
            Tree t = new Tree();
            AI a = new AI();
            Move currentMove = new Move();
            b.PrintToScreen();
            int playCounter = 0;

            // Keep playing till legal moves possible
            while (b.DidAnyoneWin() == Player.EMPTY)
            {
                if (b.turnToPlay == Player.WHITE)
                {
                    // Play randomly for WHITE
                    currentMove = a.RandomMoveMCTS(b, b.turnToPlay, t);
                    //currentMove = a.RandomMove(b, b.turnToPlay);
                }
                else if (b.turnToPlay == Player.BLACK) // BLACK plays first
                {
                    // Play the MiniMax Algorithm for BLACK
                    //currentMove = a.MiniMaxMove(b, b.turnToPlay);
                    // Play the Alpha-Beta Pruning Algorithm for BLACK
                    //currentMove = a.AlphaBetaPruningMove(b, b.turnToPlay, 1000, -1000);
                    // Play the UpperConfidenceBounds Algorithm for BLACK
                    currentMove = a.UpperConfidenceBoundsMove(b, b.turnToPlay, t);
                    // Play the MonteCarloTreeSearch Algorithm for BLACK
                    //currentMove = a.MonteCarloTreeSearchMove(b, b.turnToPlay, t);
                    //currentMove = a.RandomMove(b, b.turnToPlay);
                    // Play randomly for BLACK
                    //currentMove = a.RandomMoveMCTS(b, b.turnToPlay, t);
                }
            }
        }
    }
}
playCounter++;
    Console.WriteLine("Play Counter = " + playCounter);
    Console.WriteLine("\n");
    if (b.turnToPlay == Player.BLACK) Console.Write("Black moves from ");
    else if (b.turnToPlay == Player.WHITE) Console.Write("White moves from ");
    Console.Write("[" + currentMove.fromX + "," + currentMove.fromY + "] to [" + currentMove.toX + "," + currentMove.toY + "]");
    Console.WriteLine("\n");
    b.PlayThisMove(currentMove);
    b.PrintToScreen();

    // Add the move played to the tree and move the gamePtr
    Node[] childNodes = t.GetAllChildNodes(t.gamePtr);
    for (int n = 0; n < childNodes.Length; n++)
    {
        if (childNodes[n].move.fromX == currentMove.fromX &&
            childNodes[n].move.fromY == currentMove.fromY &&
            childNodes[n].move.toX == currentMove.toX &&
            childNodes[n].move.toY == currentMove.toY)
        {
            t.gamePtr = childNodes[n];
            childNodes[n].wasThisPlayed = true;
        }
    }
    if (b.DidAnyoneWin() == Player.BLACK)
    {
        Console.WriteLine("\n BLACK Wins !!!\n");
        Console.ReadKey();
        return;
    }
    else if (b.DidAnyoneWin() == Player.WHITE)
    {
        Console.WriteLine("\n WHITE Wins !!!\n");
        Console.ReadKey();
        return;
    
```