GAMING ALGORITHMS APPLIED TO EINSTEIN WÜRFELT NICHT!

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

By

Vivek Gulati

December, 2012
The graduate project of Vivek Gulati is approved by:

________________________________________  ______________________
Peter Gabrovsky, PhD  Date

________________________________________  ______________________
John Noga, PhD  Date

________________________________________  ______________________
Richard Lorentz, Ph.D, Chair  Date

California State University, Northridge
ACKNOWLEDGMENTS

I would like to thank Professor Richard Lorentz for all the help he has extended during this project. His continuous guidance and direction has made this project possible.
LIST OF TABLES

TABLE 1: RESULTS OF RANDOM PLAYING ................................................................. 22
TABLE 2: RESULTS OF PLAYING AS MINIMAX WITH ALPHABETA PRUNING .............. 23
TABLE 3: RESULTS OF PLAYING AS MCTS ................................................................. 23
TABLE 4: RESULTS OF PLAYING AS MCTS WITH UCT ........................................ 24
TABLE 5: RESULTS OF PLAYING ON LITTLEGOLEM AS MCTS WITH UCT ............... 24
LIST OF FIGURES

FIGURE 1: ROLLINGSTONE DESKTOP VERSION ................................................................. 2
FIGURE 2: ROLLINGSTONE WEB VERSION ............................................................................. 3
FIGURE 3: ROLLINGSTONE CONSOLE VERSION ................................................................. 4
FIGURE 4: ROLLINGSTONE ARCHITECTURE .......................................................................... 5
FIGURE 5: ASSIGNING VALUES TO NODES IN A MINIMAX TREE [13] ................................. 8
FIGURE 6: POSITIONAL VALUE FOR THE BOARD .............................................................. 11
FIGURE 8: CLASS DIAGRAM FOR THE TREE NODE .......................................................... 16
FIGURE 9: TREE STRUCTURE IN ROLLINGSTONE ............................................................. 17
FIGURE 10: TEST SETUP ..................................................................................................... 21
ABSTRACT

GAMING ALGORITHMS APPLIED TO EINSTEIN WÜRFELT NICHT!

By

Vivek Gulati

Master of Science in Computer Science

In this project I have examined a few gaming algorithms as applied to the game of EWN. The EWN is different from other games because of randomness used at two places. First while setting the board the pieces are arranged randomly and second because of the die when making a move. The algorithm that has proved to be effective is the Monte Carlo method. This paper also examines the effect of making selection using the UCT. The UCT selection must be customized depending on the nature of the game. To find an optimum UCT value I have created a computer game RollingStone that plays a large number of games against OneStone (another EWN game). RollingStone also has a web version that has been used to play on the littlegolem.net gaming website. The results of these games and the values for the UCT have been published as part of this paper.
1. INTRODUCTION

1.1 EinStein würfelt nicht! (EWN)

EWN is a board game, designed by Ingo Althöfer, a professor of applied mathematics at Friedrich Schiller University of Jena (FSU), Jena, Germany. It was the official game of an exhibition about Einstein in Germany during the Einstein Year (2005) [5].

The game board is 5×5 square fields large. Each player controls 6 playing pieces, which are stones with the numbers 1 to 6 marked on them. The stones of player 1 are set up in the six fields of the upper left corner of the board and those of player 2 in the six fields of the lower right corner of the board so they face each other diagonally. During his turn a player rolls a die. If he still has the stone with the number corresponding to the rolled value on the board he must move this stone. If the corresponding stone was already removed from play (see below) the player may choose to either move the stone with the next higher or the next lower number. In his move player 1 may move the stone one field to the right, downward or diagonally to the lower right. Respectively player 2 may move the stone one field to the left, upward or diagonally to the upper left. If a player ends his move on a field occupied by another stone, be it one of the opponent’s or one of his own stones, then this stone is removed from play. To win the game a player must occupy the corner field of the opponent with a playing piece. A player also wins if there are no opposing pieces left on the board.

1.2 Computer Program

For the purpose of this research I have written a computer program called RollingStone that implements the EWN game using different algorithms. The algorithms used in this research are:
• Random
• Minimax with Alpha-Beta pruning
• Monte-Carlo tree search

The game has three versions:

• **Desktop version**: to play against human opponents and to play in auto mode against itself using any of the above algorithms. This version of the program, because of the aid in visualization it provides, was mainly used to test the implementation and debugging of the software. The color choice and the type of player can be chosen from the left panel on the screen. Manual option can be selected for a human player. For the game to play in auto any of the algorithm options other than manual can be selected.

![RollingStone Desktop Version](image)

**Figure 1: RollingStone Desktop Version**
- **Web version**: to play on the website [www.LittleGolem.net](http://www.LittleGolem.net) against a bot or a human player from all over the world. This version was used to measure up RollingStone against real world players. Since each game takes a long time to finish this version could only play a limited number of games. The web version is mainly designed to play in the auto version but that can be overridden anytime by unchecking the AutoPlay option provided on the UI. The human player then has to type in the move to send to the opponent.

![Figure 2: RollingStone Web Version](image)

- **Console version**: this version is customized to play against OneStone, a bot designed to play EWN using MCTS. This version of RollingStone can play a large number of games in a short time. The console version of RollingStone was the one that was mainly used to collect statistics for this study. This version has a configuration file written in xml. This file can be used to choose various options
like stone color, first to move or second to move, which algorithm to play against OneStone. This config file can also be used to choose the number of games to be played. The figure below shows a game with OneStone in progress. RollingStone is playing black and sends the best move to OneStone after evaluation. OneStone is playing white and sends back the move to RollingStone.

Figure 3: RollingStone Console Version
Figure 4: RollingStone Architecture

**Problem Statement:** How MCTS compares with other algorithms?

Rollingstone was played against OneStone using different algorithms and the results were compared. The random nature of the game because of die being thrown before every move makes it difficult to write a winning algorithm. MCTS has been successfully applied in the difficult problem of computer Go. MCTS can be used with little or no domain knowledge, and has succeeded on difficult problems where other techniques have failed [3]. There are various versions of MCTS selection. One such version Upper Confidence Bound applied to Trees (UCT) is used in this project. Other algorithms are compared against the UCT version of MCTS.
**Research Question:** How to improve MCTS for the EWN game?

The MCTS requires playing a large number of random games. A move is selected after the results of these simulated games are available. In EWN there could be up to 6 moves per turn. If a move is chosen at random there is a possibility of choosing best and worst move equally. So to choose a winning move MCTS creates a tree in memory and the algorithm traverses the part of the tree that is visited more often and would then try to select the node with best win ratio. So a number of parameters like how many games to simulate or how long the simulation should run affect the selection of the move. These parameters must be adjusted based on the experiments to produce the best results. The UCT function referred in 2.3.1 has a constant that is to be fine-tuned for the problem at hand. For this project I have experimented with different values of C to get the best result.
2. ALGORITHMS

2.1 Random

The EWN game has lot of randomization built in. First the stones are set on the board in a random manner and then the die introduces another level of randomization. The random algorithm implemented for the game creates a baseline against which the other algorithms are measured. With each die throw all the legal stones that can be moved are calculated and then a list of all the moves that these stones could play is prepared. Then one of the possible legal moves is selected at random. The maximum number of possible legal moves could be 6 and the minimum could be 1. In the worst case scenario the possibility of picking a winning move is 1/6. Therefore, the worst case probability of winning the game \( = \left(\frac{1}{6}\right)^N \) where, N is the number of turns to finish the game. This is the worst case scenario; the test results show a better performance than this.

2.2 Minimax and Alpha-Beta pruning

Minimax is the simplest of the AI techniques for game playing. It involves playing out all the hypothetical moves and then choosing the most favorable move that would lead to winning the game (If the game cannot be won from the position still the algorithm plays till the end choosing a move based on the rank. Surrender has not been built into the computer program). Minimax is a decision rule used for minimizing the possible loss for a worst case (maximum loss) scenario. Minimax uses the fact that the two players are working towards opposite goals. The logic behind minimax is that the opponent will be trying to minimize whatever value or the rank the algorithm is trying to maximize. Thus the objective is to make a move that leaves its opponent capable of doing the minimum damage. For a game
like EWN there are only two possible values win or lose. That can be assigned numeric values. Once the desired depth of the tree has been traversed the algorithm then will evaluate a value from the game’s current state and assign those values to the leaf nodes. The computer then evaluates starting from the bottom of the tree which possible value is best for the opponent.

![Minimax Tree](image)

**Figure 5: Assigning values to nodes in a minimax tree [13]**

In the above tree the computer plays at levels A and C and opponent has the turn at level B. The computer is trying to maximize its score so at C1 the computer will choose the position with value 5. Similarly it will choose positions with values 11 and 8 at C2 and C3 respectively. But when the opponent plays at B it is going to choose the position that leads to minimum values at level C. Thus B1 will have a score of the minimum of 5, 11 and 8 that is 5. Similarly, the value at B2 and B3 are 2 and 3 respectively. At A again its computer’s turn to choose max of 5, 2 and 3 that is 5. This approach when applied to minimax leads to a pessimistic algorithm and most of the time it sacrifices major winnings because it assumes that the opponent is going to choose the best move. When in fact the move opponent is going to choose depends on the random die value. The EWN minimax
tree becomes much wider due to 6 die values at each node. The computer chooses a move to minimize the loss based on these 6 die values. In fact when the opponent rolls the die the value could be different from the die value attached to the move played by the computer. This pessimistic approach that makes the computer choose a defensive move also prevents it from playing a winning move. The best example of this is the minimax implementation of tic-tac-toe. With the minimax implementation the game will always end in a draw.

However, only simple games like tic-tac-toe can have their full tree generated in reasonable time. So some kind of optimization needs to be added to the algorithm. The standard way of applying optimization to minimax is to use an evaluation function. Minimax when applied with an evaluation function which evaluates the strength of each position from a certain point in game can still be useful. Evaluation function evaluates the current game position from the point of view of one player. It does this by giving a value to the current state of the game by counting the number of pieces in the board and the number of moves left to the end of the game.

Pseudo code for the minimax function (Appendix A)

function MINIMAX(N)
    if N is deep enough then
        return the estimated score of this leaf
    else
        Let N1, N2, .., Nm be the children of N;
        if N is a Min node then
            return min{MINIMAX(N1), .., MINIMAX(Nm)}
        else
            return max{MINIMAX(N1), .., MINIMAX(Nm)}
Alpha-beta pruning is an improvement over the minimax algorithm. It is possible to compute the correct minimax decision without looking at every node in the tree. Eliminating possibilities from consideration without having to examine them, the algorithm allows us to discard large parts of the tree from consideration. Consider a node \( n \) somewhere in the tree, such that one can move to that node. If there is a better choice \( m \) either at the parent of the node \( n \) or at any choice point further up, \( n \) will never be reached. Hence, alpha-beta pruning when applied to a minimax tree it returns the same move as the standard minimax but prunes away branches or sub-trees that cannot potentially influence the decision.

The evaluation function that has been chosen for this thesis is based on assigning rank to each position on the board based on its distance from the winning position. For this game each position on the board is assigned a value. The opposite corner is assigned the highest value of 100 and own corner is 10. The total rank of a leaf is the highest positional value on the board of a player subtracted by the highest positional value of the opponent’s piece. The number of pieces remaining on the board is also taken into account. The number of opponent’s pieces is subtracted from the number of pieces remaining for the player. This number is then multiplied by 5 and added to the rank. Figure below shows the positional values for a player who begins with the top left of the board.

The depth of the tree to be traversed for minimax was based on experiments. I found that to return in reasonable time of about 60 seconds the depth of tree should be 5. The other values experimented with were 3 returned in about 0.15 second, 4 returned in about 2 to 4 seconds, 5 returned in about 50 – 60 seconds and 6 returned in about 30 min.
Position rank = Highest Value (B) – Highest Value (W) + (N_B – N_W) * 5

Where, N is the number of pieces remaining on the board.

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>60</td>
<td>60</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>60</td>
<td>80</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>60</td>
<td>80</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6: Positional Value for the board**

Due to the random nature of the game the minimax has a very limited performance when applied to EWN. The children are evaluated for all the die values and then minimax value is chosen out of all these children. Writing the right evaluation function can also pose challenges. It requires much deeper knowledge of the nature of the game and also a system of ranking the positions on the board.

### 2.3 Monte Carlo Tree Search

As already mentioned the random nature of the game makes it very difficult to apply a deterministic algorithm to EWN. There is a class of algorithms that are based on repeated random sampling to compute the results called as Monte Carlo methods. These algorithms work by generating very large samples upon which the results are then based.

In this method a search tree is built according to the outcomes of simulated playouts. MCTS algorithms are generally a 4 step process that is repeated until some limit is reached, usually a limit on elapsed time or number of simulations. In this project I started with
number of simulations as the limit but then found that a limit on time yields better results. The current version of the game uses a 10 seconds limit. The steps of the algorithm, illustrated in Figure 1, are:

1) **Selection**: The algorithm selects a child node of the position currently being considered, repeating this process until a leaf node is reached. Selection balances the exploitation of known good nodes with the exploration of nodes whose value is currently uncertain.

2) **Expansion**: One or more children are added to the leaf node reached in the selection step.

3) **Simulation**: A simulation is carried out from the new leaf node, using a random move generator or other approach at each step, until a result is reached.

4) **Backpropagation**: The result at the simulation step is propagated back to all nodes in the tree that were part of the selection process to update the values (e.g. number of wins) in those nodes.
The algorithm has two principal advantages over minimax with alpha-beta pruning:

1) The algorithm can be stopped at any point to yield a result. There is no need to reach a particular stage during search, before a result is obtainable, as there would be for minimax search. The tree is built using playouts which ensures all values are always up-to-date following every iteration of the algorithm.

2) An evaluation function is not required for non-terminal game states, as simulation always reaches a terminal state.
2.3.1 Upper Confidence Bounds for Trees (UCT)

The UCT-method (which stands for Upper Confidence bounds applied to Trees) is a very natural extension to MCTS, where for each played game the first moves are selected by searching the tree, and as soon as a terminal node is found a new move/child is added to the tree and the rest of the game is played randomly. The evaluation of the finished random game is then used to update the statistics of all moves in the tree that were part of that game. UCT also selects the best move most of the time but also explores other moves in a more sophisticated way. It does this by adding a number to the winrate (wins/visits) for each candidate move that goes down every time the node has been visited. But this number also goes up a little every time the parent was visited and some other move is selected. This means that the win rate + the number will grow for unexplored moves so that at some point the sum is higher than for all moves that have higher winrates. If the move is a winning move, then the winrate goes up and it may soon be selected again. If the move failed to win then the winrate goes down as well as the added number and the move may have to wait for a long time before the added number has grown large enough. A move can also be selected if all other moves are refuted so that the winrates for all competitors goes down. Starting from the root, UCT searches a path of moves through the tree by calculating a value for each candidate position according to the rate of win and how many times the position has been played as well as how many times the position has been visited. If there are children to a node that have not been visited then one of those moves are selected randomly. With UCT one can strike a good balance between searching the best move so far (exploiting) and exploring alternative moves [12].
Node selection during tree descent is achieved by choosing the node that maximizes some quantity. An Upper Confidence Bounds (UCB) formula of the following form is typically used:

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

Where \( v_i \) is the wins/visits value of the node, \( n_i \) is the number of the times the node has been visited and \( N \) is the total number of times that its parent has been visited. \( C \) is a tunable bias parameter.
3. ROLLINGSTONE - IMPLEMENTATION OF MCTS

3.1 Tree Structure

Implementation of MCTS has to be customized for each game. At the heart of MCTS lies a tree that keeps track of the moves. RollingStone starts out with a root node that has all the possible legal moves at level 1. Each of these moves will have children that are the possible moves for each die position 1 through 6 for the other player. The next level is again for the alternate color for each die position. This way the tree keeps growing.

3.1.1 Tree Class

Node class has a 2 dimensional array of Node as class member to store the child nodes. The first dimension is of size 6 depicting each die value. The second dimension is variable representing the number of legal moves for that die value. Other class members are:

Move: that stores the move represented by the child node.
Parent: is the pointer to the parent node
Sibling: pointer to the other nodes that have the same parent
Color: color of the stone

**Figure 8: Class diagram for the tree node**

Node class has a 2 dimensional array of Node as class member to store the child nodes. The first dimension is of size 6 depicting each die value. The second dimension is variable representing the number of legal moves for that die value. Other class members are:

Move: that stores the move represented by the child node.
Parent: is the pointer to the parent node
Sibling: pointer to the other nodes that have the same parent
Color: color of the stone
Visitsnum: keeps track of the number of times this node has been visited

Wins: keeps track of the number of the wins for the node

![Tree structure in RollingStone](image)

**Figure 9: Tree structure in RollingStone**

### 3.2 Selection

The C in the equation in section 2.3.1 is a constant and has to be tuned to get the best results from the MCTS. Each game and each game implementation has to find its own value of C depending on the experiments. Fine tuning the values based on experiments is a
time consuming task because a large number of trials need to be run for the purpose. In many domains it has proved beneficial to influence the score for each action using domain knowledge, to bias the search towards/away from certain actions and make use of other forms of reward estimate. [3]

For my experiments I tried different values of C ranging from 0.1 to 100. Initially I set the C=0 which is a special case of the UCT equation that simply results in the win/visits ratio. The node with the highest win percentage is selected. The results of this selection are recorded in the next section. For the next set of results the C was set at 0.1, 1, $\sqrt{2}$, 10 and 100 the results from these values of C are recorded in the test results section.

### 3.3 Expansion

During the expansion phase child nodes are created. The number of children depends on the number of possible legal moves for each die position. The implementation has a two dimensional array of child nodes as a class member of each node. The first dimension of this array is fixed to 6 representing each die position. The second dimension is variable depending on the possible legal moves for each die value. The number of legal moves and

<table>
<thead>
<tr>
<th>Pseudo code for the UCT selection GetUCTMove (Appendix A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BestChild = null;</td>
</tr>
<tr>
<td>foreach n[i] = parent.child[i]</td>
</tr>
<tr>
<td>if (BestChild == null</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>BestChild = n[i];</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>Function: UCTValue(child n[i])</td>
</tr>
<tr>
<td>UCTValue = n.wins / n.visitsnum) +</td>
</tr>
<tr>
<td>C* Math.Sqrt(Math.Log(n.parent.visitsnum)/n.visitsnum);</td>
</tr>
<tr>
<td>Return UCTValue;</td>
</tr>
</tbody>
</table>
hence children for each die value could be from 1 through 6. This is also depicted in Figure 9 above.

Pseudo code for the createchildren function (Appendix A)

```plaintext
for (die = 1 to 6) {
    moves[] = AllPossibleMoves(die)
    createchildren(parent, moves[], die);
}
Function: createchildren(parent, moves[],die) {
    if (parent.child[die - 1] == null) {
        for all moves
            create parent.child
    }
    return parent.child;
}
```

### 3.4 Simulation

Once the nodes have been added in the expansion phase a node is picked at random to play a simulated game. There are several enhancements that were tried in this phase. Instead of picking a random move a bias factor can be applied that prefers certain move over others. For instance a move closer to the win was given preference over the other moves. A diagonal move that would reach the goal faster was given preference. On the other hand if the number of opponent’s stones is 3 or less then any move (other than the winning move) which results in removing of one of these stones is avoided.
3.5 BackPropagation

Once the simulation is complete the back propagation is applied to update the nodes. The leaf node backtracks to the root node and increments the win count for the player color that won the game.

Pseudo code for the simulation phase (Appendix A)

Nodes = GetAlltheChildNodes;
Array DiagonalMoves;
For each node
  If(node.move is winning move)
    Play this move
  Else if (node.move is a diagonal move)
    Check if bringing down opponent count to less than 2
  Else
    Add(DiagonalMoves)

  If(DiagonalMoves.count>0)
    Select random and play the move
  Else
    Select random move and play

Pseudo code for the back propagation phase (Appendix A)

If(board.winner == p)
  Node.win++;
While(node.parent != null)
  {
    If(node.parent.color == p)
    {
      Node.parent.win++;
    }
    Node = node.parent;
  }
4. TEST ENVIRONMENT

The tests were carried on Intel i7 machine using 6GB of memory on Windows7 platform. The computer program has been written in C# and it communicates with the OneStone program, written in C++, using named pipes. The objective of this test was to compare the results from running algorithms for the EWN game and also to find the optimum parameters for the UCT function.

![Diagram showing the test setup](image)

**Figure 10: Test Setup**

4.1 Random

The first tests were done using the Random move selection. The purpose of this test was to establish a baseline. One move was selected at random from the possible moves.
RollingStone when played in random mode with OneStone won about 7.5% of the times when playing first. The winning percentage went down to about 4.5% when played as second. The maximum number of moves available at a time could be 6. That means the probability of picking the best move for the worst case scenario is 1/6. The chances of winning the game diminish with every iteration. Therefore to reduce the number of moves to finish the game one small variation was added that was to give priority to diagonal moves when available. The results after this variation were still similar.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Configuration</th>
<th>Opponent</th>
<th># Games</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>Black, 1st Move</td>
<td>OneStone</td>
<td>200</td>
<td>15</td>
</tr>
<tr>
<td>Random</td>
<td>White, 2nd Move</td>
<td>OneStone</td>
<td>200</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1: Results of random playing

4.2 AlphaBeta

The next set of experiment was done with minimax. The challenge with minimax was to choose the right evaluation function

When played with Minimax with alphabeta pruning RollingStone won about 18% of the games while playing first. When playing second it won about 22% of the games. The minimax algorithm though performs better than the random it is still not very effective in coming up with a winning player bot. Some more advanced methods would need to be put in place to get better results.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Configuration</th>
<th>Opponent</th>
<th># Games</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaBeta</td>
<td>Black, 1st Move</td>
<td>OneStone</td>
<td>200</td>
<td>36</td>
</tr>
<tr>
<td>AlphaBeta</td>
<td>White, 2nd Move</td>
<td>OneStone</td>
<td>200</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 2: Results of playing as Minimax with alphabeta pruning

4.3 MCTS

MCTS with a simple selection method of picking the move that has the maximum wins per visit is a special case of the UCT selection where C=0. The games played with C=0 improves the win percentage significantly over minimax. RollingStone player wins about 41.5% of games when playing first. The percentage of wins with MCTS is almost double that of the alphabeta. The winning percentage while playing second drops to 41% but still the performance is better than that of alphabeta.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Configuration</th>
<th>Opponent</th>
<th># Games</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCTS</td>
<td>Black, 1st Move</td>
<td>OneStone</td>
<td>200</td>
<td>83</td>
</tr>
<tr>
<td>MCTS</td>
<td>White, 2nd Move</td>
<td>OneStone</td>
<td>100</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 3: Results of playing as MCTS

4.4 MCTS with UCT selection

Improvements over the previous run of MCTS were applied; the selection was changed to UCT with a value of 1 and the results showed improvement over MCTS. Other values of C
were experimented with ranging from 0.1 to 100 but the value of 1 yielded the best results. These have been recorded in the tables below. With C=1 MCTS with UCT as the selection method yields a win percentage of about 51% when playing first and a 48.5% when playing as second. The UCT is able to produce better results than the simple selection.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>C</th>
<th>Configuration</th>
<th>Opponent</th>
<th># Games</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT</td>
<td>1</td>
<td>Black, 1st Move</td>
<td>OneStone</td>
<td>150</td>
<td>77</td>
</tr>
<tr>
<td>UCT</td>
<td>1</td>
<td>White, 2nd Move</td>
<td>OneStone</td>
<td>150</td>
<td>73</td>
</tr>
<tr>
<td>UCT</td>
<td>0.1</td>
<td>Black, 1st Move</td>
<td>OneStone</td>
<td>100</td>
<td>35</td>
</tr>
<tr>
<td>UCT</td>
<td>10</td>
<td>Black, 1st Move</td>
<td>OneStone</td>
<td>100</td>
<td>31</td>
</tr>
<tr>
<td>UCT</td>
<td>100</td>
<td>Black, 1st Move</td>
<td>OneStone</td>
<td>100</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 4: Results of playing as MCTS with UCT

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Configuration</th>
<th>Opponent</th>
<th># Games</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCT</td>
<td>Any</td>
<td>LittleGolem</td>
<td>30</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 5: Results of playing on LittleGolem as MCTS with UCT
5. CONCLUSION

*How MCTS compares with other algorithms?*

When choosing a gaming algorithm the choice between Minimax and MCTS may be difficult. For a game like EWN where possibilities lead to a nontrivial tree the results of the minimax have limited success. Another factor that contributes to this is the domain knowledge of the game. Unable to write an effective evaluation function can make Minimax unsuitable for a game like EWN. MCTS approaches to game such as Chess are not as successful as for the game of Go. In conclusion if domain-specific knowledge is readily available then both the algorithms may be viable.

*How to improve MCTS for the EWN game?*

As shown in the results in the previous section the best results are obtained by using UCT selection and a combination of game specific improvements. The game specific enhancements play a major role in making the implementation superior than the versions without these enhancements. Also the value for C = 1 produced the best results with UCT selection.
6. FUTURE RESEARCH

The game specific enhancements introduced are not comprehensive by any means. There are many more enhancements that can be applied to the game. Also the enhancements already applied have inbuilt shortcomings. If these enhancements strengthen a particular aspect of the game at the same time they weaken others. For example a move that brings down the opponent stone count below 3 is avoided. That means RollingStone is not even targeting winning the game by capturing all of the opponents stones. These blindsided weaknesses will be mitigated as part of the future research. Another topic for future research is to build a database of the board configuration and the moves. Storing all the possible board configurations is not possible. But when the number of stones on the board is down to a manageable number say 6 there is still lot of game remaining and having a database of winning moves for these configurations will definitely strengthen the game. There is a player Sybil_c on littlegolem that has a perfect database for the moves when 7 stones are remaining on the board. Currently, on one machine the game can play only one instance when playing against OneStone. This limits the throughput. The computer program will be enhanced to play multiple instances.
REFERENCES


   http://scim.brad.ac.uk/staff/pdf/picowlin/CowlingWardPowley2012.pdf [10/02/2012]


4. Monte-Carlo Tree Search
   http://www.unimaas.nl/games/files/phd/Chaslot_thesis.pdf [02/01/2012]


6. www.littergolem.net [02/01/2012]

7. www.mastermoves.eu [02/01/2012]


10. The *-Minimax Search Procedure for Trees Containing Chance Nodes, Bruce W. Ballard,Department of Computer Science, Duke University, Durham, NC

11. The UCT Algorithm Applied to Games with Imperfect Information (http://www.b0n541.net/study/jan_schaefer_uct_algorithm_applied_to_games_with_imperfect_information.pdf) [11/30/2012]


14. The Minimax Algorithm, Glenn Strong
using System;
using System.Collections.Generic;
using System.Text;

namespace EWN
{
    public enum Player { W = 1, B = 2, Open = 3 }
    public enum Stones { W1 = 11, W2 = 12, W3 = 13, W4 = 14, W5 = 15, W6 = 16, B1 = 21, B2 = 22, B3 = 23, B4 = 24, B5 = 25, B6 = 26, OO = 0 }
    public class Space
    {
        public int X;
        public int Y;
        public Space(int x, int y)
        {
        }
    }
}
this.X = x;
this.Y = y;
}
public Space(Space space)
{
    this.X = space.X;
    this.Y = space.Y;
}

public struct Move
{
    public Space From;
    public Space To;
    public int Rank;
    public Move(Space from, Space to)
    {
        From = new Space(from);
        To = new Space(to);
        Rank = 0;
    }
}

public class EWNBoard
{
    public const int boardsize = 5;
    public Stones[,] Board;
    public EWNBoard()
    {
    }
    public EWNBoard(bool newgame)
    {
        Board = new Stones[boardsize, boardsize] {
            {Stones.OO, Stones.OO, Stones.OO, Stones.OO, },
            {Stones.OO, Stones.OO, Stones.OO, Stones.OO, },
            {Stones.OO, Stones.OO, Stones.OO, Stones.OO, },
            {Stones.OO, Stones.OO, Stones.OO, Stones.OO, },
            {Stones.OO, Stones.OO, Stones.OO, Stones.OO, }
        };
        if (newgame)
        {
            Arrangenewboard();
        }
    }
}

private void Arrangenewboard()
{
    Random r = new Random();
    //Arrange Black
for (int i = 0; i <= 2; i++)
{
    for (int j = 0; j <= 2 - i; j++)
    {
        int count = r.Next(0, blackstones.Count);
        Board[i, j] = blackstones[count];
        blackstones.RemoveAt(count);
    }
}
for (int i = 4; i >= 2; i--)
{
    for (int j = 4; j >= 6 - i; j--)
    {
        int count = r.Next(0, whitestones.Count);
        Board[i, j] = whitestones[count];
        whitestones.RemoveAt(count);
    }
}

public EWNBoard(string White, string Black)
{
    Board = new Stones[boardsize, boardsize] {
    };
    White = White.Trim();
    Black = Black.Trim();
    for (int i = 0; i <= 2; i++)
    {
        for (int j = 0; j <= 2 - i; j++)
        {
            Board[i, j] =
            blackstones[int.Parse(Black[0].ToString()) - 1];
            Black = Black.Remove(0, 1);
        }
    }
    for (int i = 4; i >= 2; i--)
    {
        for (int j = 4; j >= 6 - i; j--)
        {
            Board[i, j] =
            whitestones[int.Parse(White[0].ToString()) - 1];
            White = White.Remove(0, 1);
        }
    }
}
```csharp
public Move[] GetAllMoves(Player p, int die)
{
    List<Move> legalmoves = new List<Move>();
    Space[] from = GetAllStones(p, die);
    if (from.Length > 0)
    {
        foreach (Space s in from)
        {
            Move[] m = GetMoves(p, s);
            if (m.Length > 0)
            {
                legalmoves.AddRange(m);
            }
        }
    }
    else
    {
        Console.WriteLine("GetAllMoves I shouldn't be here");
        //no legal pieces left. Game Over.
        //Throw exception this shouldn't be called
    }
    return legalmoves.ToArray();
}

public bool EndGame
{
    get
    {
        if (Winner == Player.Open)
            return false;
        else
            return true;
    }
}

public Player Winner
{
    get
    {
        string boardstones = "";
        //black at position 44
        for (int i = 0; i < boardsize; i++)
        {
            for (int j = 0; j < boardsize; j++)
            {
                boardstones += Board[i, j].ToString();
            }
        }

        if (Board[4, 4].ToString().IndexOf("B") != -1 ||
            boardstones.IndexOf("W") == -1)
        {
            return Player.B;
        }
        if (Board[0, 0].ToString().IndexOf("W") != -1 ||
            boardstones.IndexOf("B") == -1)
        {
```
return Player.W;
}
return Player.Open;
}

public Player PlayeratPosition(Space s)
{
    Player p = Player.Open;
    if (Board[s.X, s.Y].ToString().IndexOf("W") != -1)
    {
        p = Player.W;
    }
    else if (Board[s.X, s.Y].ToString().IndexOf("B") != -1)
    {
        p = Player.B;
    }
    return p;
}

public void Move(Move move)
{
}

public Move[] GetMoves(Player p, Space from)
{
    List&lt;Move&gt; moves = new List&lt;Move&gt;();
    if (p == Player.B)
    {
        if (from.X + 1 < boardsize)
        {
            moves.Add(new Move(from, new Space(from.X + 1, from.Y)));
        }
        if (from.Y + 1 < boardsize)
        {
            moves.Add(new Move(from, new Space(from.X, from.Y + 1)));
        }
        if (from.X + 1 < boardsize && from.Y + 1 < boardsize)
        {
            moves.Add(new Move(from, new Space(from.X + 1, from.Y + 1)));
        }
    }
    else if (p == Player.W)
    {
        if (from.X - 1 >= 0)
        {
            moves.Add(new Move(from, new Space(from.X - 1, from.Y)));
        }
        if (from.Y - 1 >= 0)
        {
            moves.Add(new Move(from, new Space(from.X, from.Y - 1));
        }
        if (from.X + 1 < boardsize && from.Y - 1 < boardsize)
        {
            moves.Add(new Move(from, new Space(from.X + 1, from.Y - 1)));
        }
    }
    return moves.ToArray();
moves.Add(new Move(from, new Space(from.X, from.Y - 1)));
} else
{
    Console.WriteLine("GetMoves I shouldn't be here");

    //throw exception. Shouldn't come here
    return moves.ToArray();
}

private Space GetStone(Player p, int die)
{
    Space s = null;
    for (int i = 0; i < boardsize; i++)
    {
        for (int j = 0; j < boardsize; j++)
        {
            if (Board[i, j] == (Stones)((int)p * 10 + die))
            {
                s = new Space(i, j);
                return (Space)s; //return here to cut down
            }
        }
    }
    return (Space)s;
}

public Space[] GetAllStones(Player p, int die)
{
    Space s = GetStone(p, die);
    if (s != null)
    {
        return new Space[] { (Space)s };  
    }
    else
    {
        //find upper and lower stones
        List<Space> legalstones = new List<Space>();
        for (int i = die + 1; i <= 6; i++)
        {
            s = GetStone(p, i);
            if (s != null)
            {
                legalstones.Add((Space)s);
                break;
            }
        }
        for (int i = die - 1; i >= 1; i--)
        {
            s = GetStone(p, i);
        }
    }
}
if (s != null)
{
    legalstones.Add((Space)s);
    break;
}
}
return legalstones.ToArray();
}
public void PrintBoard()
{
    for (int i = 0; i < boardsize; i++)
    {
        Console.WriteLine("---------------------");
        Console.Write("|");
        for (int j = 0; j < boardsize; j++)
        {
            Console.Write((Board[i, j] == Stones.OO ? "   " : " 
 + Board[i, j].ToString()) + "|"));
        } Console.WriteLine("\n");
    } Console.WriteLine("---------------------");
} public int Count(Player p)
{
    int count = 0;
    for (int i = 0; i < boardsize; i++)
    {
        for (int j = 0; j < boardsize; j++)
        {
            if ((Board[i, j].ToString()).IndexOf(p.ToString()) != -1)
            count++;
        }
    } return count;
}
public EWNBoard Clone()
{
    EWNBoard b = new EWNBoard();
    b.Board = (Stones[,] this.Board.Clone();
    return b;
}
}
using System;
using System.Collections.Generic;
using System.Text;

namespace EWN
{
    public class AI
    {

        public static int[,] PositionRankWhite = new int[,]{
            {-100, -80, -60, -20, -10},
            {-80, -80, -60, -20, -10},
            {-60, -60, -60, -20, -10},
            {-20, -20, -20, -20, -10},
            {-10, -10, -10, -10, -10}};

        public static int[,] PositionRankBlack = new int[,]{
            {10, 10, 10, 10, 10},
            {10, 20, 20, 20, 20},
            {10, 20, 60, 60, 60},
            {10, 20, 60, 80, 80},
            {10, 20, 60, 80, 100}};

        const int timeout = 100000000;

        public static Move GetRandomMove(EWNBoard board, Player p, int die)
        {
            Move[] moves = board.GetAllMoves(p, die);
            Random r = new Random();
            // Code continues here...
        }
    }
}
```csharp
    int randomemove = r.Next(0, moves.Length);
    return moves[randomemove];
}

public static Move GetMCTSMove(EWNBoard board, Player p, int die, Tree t)
{
    Node BestChild = null;
    Move[] moves = board.GetAllMoves(p, die);
    EWNBoard newboard;
    Node[] n = t.createchildren(t.pointer, moves, die);
    for (int i = 0; i < n.Length; i++)
    {
        DateTime s1 = DateTime.Now;
        long diff = 0;
        while (diff <= timeout)
        {
            diff = DateTime.Now.Ticks - s1.Ticks;
            //Console.WriteLine("Diff = "+diff+" Time Elapsed = " + (DateTime.Now - s1));
            newboard = board.Clone();
            newboard.Move(n[i].move);
            n[i].visitsnum++;
            t.pointer = n[i];
            if (newboard.Winner == p)
            {
                n[i].wins++;
                n[i].move.Rank++;
                continue;
            }
            while (!newboard.EndGame)
            {
                int idie = Dice.ThrowDice;
                Move move = GetRandomMove(newboard, (p == Player.B) ? Player.W : Player.B), idie);
                newboard.Move(move);
                //n = t.createchildren(t.pointer, moves);
                //int randommove = new Random().Next(0, n.Length);
                //newboard.Move(n[randommove]);
                //n[randommove].visitsnum++;
                //t.pointer = n[randommove];
                if (newboard.Winner == Player.Open)
                {
                    idie = Dice.ThrowDice;
                    moves = newboard.GetAllMoves(p, idie);
                    Node[] ntemp = t.createchildren(t.pointer, moves, die);
                    int randommove = new Random().Next(0, ntemp.Length);
                    newboard.Move(ntemp[randommove].move);
                    ntemp[randommove].visitsnum++;
                    t.pointer = ntemp[randommove];
                }
            }
        }
    }
}
```
if (newboard.Winner == p)
{
    ntemp[randommove].wins++;
    Node tempnode = ntemp[randommove];
    //n[i].wins++;
    //n[i].move.Rank++;
    //reverse traverse and mark the parent win till you reach the root.
    while (tempnode.parent != null)
    {
        tempnode.parent.wins++; 
        tempnode = tempnode.parent;
    }
    break;
}

Console.WriteLine(Program.movetoboard(n[i].move, board) + " Winning chance = " + ((n[i].wins) * 100 / (n[i].visitsnum)) + "%");

//Console.WriteLine("Visits = "+n[i].visitsnum+ " Wins = "+n[i].wins+" Winning chance = " +((n[i].wins)*100/(n[i].visitsnum)) + ");
if (BestChild == null || n[i].wins > BestChild.wins)
{
    BestChild = n[i];
}

t.pointer = BestChild;
return BestChild.move;

public static Move GetUCTMove(EWNBoard board, Player p, int die, Tree t)
{
    Node BestChild = null;
    Move[] moves = board.GetAllMoves(p, die);
    EWNBoard newboard;
    Node[] n = t.createchildren(t.pointer, moves, die, p);
    for (int i = 0; i < n.Length; i++)
    {
        DateTime sl = DateTime.Now;
        long diff = 0;
        while (diff <= timeout)
        {
            diff = DateTime.Now.Ticks - sl.Ticks;
            //Console.WriteLine("Diff = "+diff+" Time Elapsed = " + (DateTime.Now - sl));
            newboard = board.Clone();
            newboard.Move(n[i].move);
            //n[i].visitsnum++;
t.pointer = n[i];
if (newboard.Winner == p)
{
    n[i].wins++;
n[i].visitsnum++;
    Node tempnode = n[i];
    //reverse traverse and mark the parent win till you reach the root.
    while (tempnode.parent != null)
    {
        if (tempnode.parent.color == p)
        {
            tempnode.parent.wins++;
        }
        tempnode.parent.visitsnum++; //check
        tempnode = tempnode.parent;
    }
    return n[i].move;
}
////////
while (!newboard.EndGame)
{
    //int idie = Dice.ThrowDice; //--uncomment to converge fast and comment below loop for 1 to 6
    //newboard.Move(move);
    EWNBoard tboard = newboard.Clone();
    for (int jdie = 1; jdie <= 6; jdie++)
    {
        newboard = tboard.Clone();
moves = newboard.GetAllMoves(p, jdie);
        int randmove = new Random().Next(0, nodetemp.Length); //replace by function selectamove()
        newboard.Move(nodetemp[randmove].move);
        t.pointer = nodetemp[randmove];
        //n = t.createchildren(t.pointer, moves);
        //int randommove = new Random().Next(0, n.Length);
        //newboard.Move(n[randommove]);
        //n[randommove].visitsnum++;
        //t.pointer = n[randommove];
        if (newboard.Winner == Player.Open)
        {
            EWNBoard tempboard = newboard.Clone();
            //idie = Dice.ThrowDice; //--uncomment
to converge fast and comment below loop for 1 to 6
            for (int idie = 1; idie <= 6; idie++)
            {
                newboard = tempboard.Clone();
            }
        }
    }
}
moves = newboard.GetAllMoves(p, idie);

Node[] ntemp = t.createchildren(t.pointer, moves, idie, p);
int randommove = new Random().Next(0, ntemp.Length);//replace by function selectamove()

newboard.Move(ntemp[randommove].move);

//ntemp[randommove].visitsnum++;
t.pointer = ntemp[randommove];
if (newboard.Winner == p)
{
    ntemp[randommove].wins++;
    ntemp[randommove].visitsnum++;
    Node tempnode = ntemp[randommove];
    //n[i].wins++;
    //n[i].move.Rank++;
    //reverse traverse and mark the parent win till you reach the root.
    while (tempnode.parent != null)
    {
        if (tempnode.parent.color == p)
        {
            tempnode.parent.wins++;
        }
        tempnode.parent.visitsnum++;
        tempnode = tempnode.parent;
        // break; -- check
    }
} 
else //other player has won
{
    //update stats
    t.pointer.visitsnum++; 
    Node tempnode = t.pointer;
    while (temnode.parent != null)
    {
        temnode.parent.visitsnum++;
        temnode = temnode.parent;
    }
}

} 
} 
} 

///// ///
Console.WriteLine(Program.movetoboard(n[i].move, board) + " Winning chance = " + UCTValue(n[i]));
```
Console.WriteLine("Visits = "+n[i].visitsnum + " Wins = "+n[i].wins + " Winning chance = "+((n[i].wins)*100/(n[i].visitsnum)) + ";
if (BestChild == null || UCTValue(n[i]) > UCTValue(BestChild))
{
    BestChild = n[i];
}
}
t.pointer = BestChild;
return BestChild.move;

static double UCTValue(Node n)
{
    double C = 1;
    double value = 0;
    value = ((double)n.wins / n.visitsnum);
    if (n.parent.visitsnum > 0)
    {
        double A = Math.Log(n.parent.visitsnum);
        double B = Math.Sqrt(A / n.visitsnum);
        value += C * B;
    }
    return value;
}
//static string movetoboard(Move move, EWNBoard board)
//{
//    string cmdstr = "-";
//    {
//        cmdstr = "x";
//    }
//}
public static Move GetMCMove(EWNBoard board, Player p, int die, int count)
{
    //Get the first level moves and then play a large number of random games for each of those moves
    Move[] moves = board.GetAllMoves(p, die);
    EWNBoard newboard;
    Move? BestMove = null;
    for (int j = 0; j < moves.Length; j++)
    {
        Move m = moves[j];
        for (int i = 0; i < count; i++)
```
newboard = board.Clone();
newboard.Move(m);
if (newboard.Winner == p)
{
    m.Rank++;
    continue;
}

while (!newboard.EndGame)
{
    int idie = Dice.ThrowDice;
    newboard.Move(move);
    if (newboard.Winner == Player.Open)
    {
        idie = Dice.ThrowDice;
        move = GetRandomMove(newboard, p, idie);
        newboard.Move(move);
        if (newboard.Winner == p)
        {
            m.Rank++;
            break;
        }
    }
}

Console.WriteLine("Winning chance = " + (m.Rank * 100 / count) + ");
if (BestMove == null || m.Rank > ((Move)(BestMove)).Rank)
{
    BestMove = m;
}
return (Move)BestMove;
}

public static Move GetMiniMaxMove(EWNBoard board, Player p, int die, int count)
{
    Move? bestmove = null;
    Move[] moves = board.GetAllMoves(p, die);
    EWNBoard newboard;
    count++;

    //Console.WriteLine(" = " + count + " ");
    for (int i = 0; i < moves.Length; i++)
    {
        newboard = board.Clone();
        Move newmove = moves[i];
        newmove.Rank = EvaluationFunction(newboard, newmove, p, die, count);
        newboard.Move(newmove);
    }

    return bestmove;
if (newboard.Winner == Player.Open)
{
    Move tempmove = GetMiniMaxMove(newboard, (p == Player.B ? Player.W : Player.B), die);
    newmove.Rank = tempmove.Rank;
}
else
{
    if (newboard.Winner == Player.B)
    {
        newmove.Rank = 100;
    }
    else if (newboard.Winner == Player.W)
    {
        newmove.Rank = -100;
    }
    else
    {
        if (newmove.Rank == 0 && newboard.Winner == Player.Open)
        {
            newmove.Rank = 0;
            Move tempmove;
            for (int idie = 1; idie <= 6; idie++)
            {
                tempmove = GetMiniMaxMove(newboard, (p == Player.B ? Player.W : Player.B), idie, count);
                if (p == Player.B)
                {
                    if (newmove.Rank < tempmove.Rank)
                        newmove.Rank = tempmove.Rank;
                }
                else
                {
                    if (newmove.Rank > tempmove.Rank)
                        newmove.Rank = tempmove.Rank;
                }
            }
        }
    }
}
{
    bestmove = newmove;
}
return (Move)bestmove;
}
private static int EvaluationFunction(EWNBoard board, Move newmove, Player p, int die, int count)
{
    int rank = 0;
    int downrankfactor = 5 * (count - 1); // the deeper the search goes lower the rank. A winning move in 2 moves should be ranked higher than the one in 3
    Player currentStone = board.PlayeratPosition(newmove.To);
EWNBoard tempboard = board.Clone();
tempboard.Move(newmove);

if (p == Player.B)
{
    if (tempboard.Winner == Player.B)
        rank = 100 - downrankfactor;
    else if (PositionRankBlack[newmove.To.X, newmove.To.Y] == 80)
        rank = 80 - downrankfactor;
    else if (currentStone != Player.Open && ((currentStone == Player.B && board.Count(p) > 3)))
        rank = 50;
        if (PositionRankBlack[newmove.To.X, newmove.To.Y] > rank)
            rank = PositionRankBlack[newmove.To.X, newmove.To.Y];
        rank = rank - downrankfactor;
    else if (count >= 5)
        rank = PositionRankBlack[newmove.To.X, newmove.To.Y];
        rank = (rank - downrankfactor) > 0 ? (rank - downrankfactor) : 5;
}
else if (p == Player.W)
{
    if (tempboard.Winner == Player.W)
        rank = -100 + downrankfactor;
    else if (PositionRankWhite[newmove.To.X, newmove.To.Y] == -80)
        rank = -80 + downrankfactor;
    else if (currentStone != Player.Open && ((currentStone == Player.W && board.Count(p) > 3)))
        rank = -50;
        if (PositionRankWhite[newmove.To.X, newmove.To.Y] < rank)
            rank = PositionRankWhite[newmove.To.X, newmove.To.Y];
        rank = rank + downrankfactor;
else if (count >= 5)
{
    rank = PositionRankWhite[newmove.To.X, newmove.To.Y];
    rank = (rank + downrankfactor) > 0 ? -5 : (rank + downrankfactor);
}

return rank;

public static Move GetMiniMaxAlphaBetaMove(EWNBoard board, Player p, int die, int count, int alpha, int beta)
{
    Move? bestmove = null;
    Move[] moves = board.GetAllMoves(p, die);
    EWNBoard newboard;
    count++;

    for (int i = 0; i < moves.Length; i++)
    {
        newboard = board.Clone();
        Move newmove = moves[i];
        newmove.Rank = EvaluationFunction(newboard, newmove, p, die, count);
        newboard.Move(newmove);

        if (newmove.Rank == 0 && newboard.Winner == Player.Open)
        {
            newmove.Rank = 0;
            Move tempmove;
            for (int idie = 1; idie <= 6; idie++)
            {
                //Have an array of 6 moves
                //populate this array for each dice value
                //Take the max (or min) of this as the rank
                if (p == Player.B)
                {
                    if (newmove.Rank == 0 || newmove.Rank < tempmove.Rank)
                    {
                        newmove.Rank = tempmove.Rank;
                    }
                }
                else
                {
                    if (newmove.Rank == 0 || newmove.Rank > tempmove.Rank)
                    {
                        newmove.Rank = tempmove.Rank;
                    }
                }
            }

            if (p == Player.B)
{  
  if (alpha < newmove.Rank)  
  {  
    alpha = newmove.Rank;  
    bestmove = newmove;  
    if (alpha >= beta)  
    {  
      break;  
    }  
  }  
  else  
  {  
    if (beta > newmove.Rank)  
    {  
      beta = newmove.Rank;  
      bestmove = newmove;  
      if (alpha >= beta)  
      {  
        break;  
      }  
    }  
    {  
      bestmove = newmove;  
    }  
  }  
  return (Move)bestmove;  
}  
[Obsolete]  
private static int EvaluationFunctionAlphaBeta(EWNBoard board, Move newmove, Player p, int die, int count, int alpha, int beta)  
{  
  int rank = 0;  
  int downrankfactor = 5 * (count - 1); // the deeper the search goes lower the rank. A winning move in 2 moves should be ranked higher than the one in 3  
  Player currentStone = board.PlayeratPosition(newmove.To);  
  EWNBoard tempboard = board.Clone();  
  tempboard.Move(newmove);  
  if (tempboard.Winner == Player.B)  
  {  
    rank = 100 - downrankfactor;  
  }  
  else if (tempboard.Winner == Player.W)  
  {  
    rank = -100 + downrankfactor;  
  }  
  else if (count >= 5)  
  {  
    if (p == Player.B)  
    {  
      /*
       *
* stones more than 3 then randomize rank = 25
* stones more than 3 and removing stone of same

```csharp
if (board.Count(p) > 3)
{
    rank = 25;
}
{
    rank = 50;
}
if (PositionRankBlack[newmove.To.X, newmove.To.Y] > rank)
{
    rank = PositionRankBlack[newmove.To.X, newmove.To.Y];
}
rank = (rank - downrankfactor) > 0 ? (rank - downrankfactor) : 5;

else if (p == Player.W)
{
    if (board.Count(p) > 3)
    {
        rank = -25;
    }
    {
        rank = -50;
    }
    if (PositionRankWhite[newmove.To.X, newmove.To.Y] < rank)
    {
        rank = -1 * PositionRankWhite[newmove.To.X, newmove.To.Y];
    }
    rank = (rank + downrankfactor) > 0 ? -5 : (rank + downrankfactor);
}
return rank;
```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.IO;
using System.Runtime.InteropServices;
using Microsoft.Win32.SafeHandles;
using System.Threading;
using System.Xml;

namespace EWN
{

class Program
{
    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern SafeFileHandle CreateFile(String pipeName, uint dwDesiredAccess, uint dwShareMode, IntPtr lpSecurityAttributes, uint dwCreationDisposition, uint dwFlagsAndAttributes, IntPtr hTemplate);

    [DllImport("kernel32.dll", SetLastError = true)]
    public static extern int ConnectNamedPipe(SafeFileHandle hNamedPipe, IntPtr lpOverlapped);
}
public const uint GENERIC_READ = (0x80000000);
public const uint GENERIC_WRITE = (0x40000000);
public const uint OPEN_EXISTING = 3;
public const uint FILE_FLAG_OVERLAPPED = (0x40000000);
public const int BUFFER_SIZE = 4096;

public static string[,] boardpositions = new string[,]{{"e1", "d1", "c1", "b1", "a1"},
{"e2", "d2", "c2", "b2", "a2"},
{"e3", "d3", "c3", "b3", "a3"},
{"e4", "d4", "c4", "b4", "a4"},
{"e5", "d5", "c5", "b5", "a5"}};

static void Main(string[] args)
{
    int GamesCount;
    bool MyFirstMove;
    Player MyColor;
    string Algorithm;

    readconfiguration(out GamesCount, out MyFirstMove, out MyColor, out Algorithm);

    DateTime startgame;
    DateTime startmove;
    DateTime endmove;
    DateTime endgame;
    EWNBoard ewn;
    int depth = 1000;
    string pipeName = "\\.\\pipe\testpipe";
    FileStream stream;
    SafeFileHandle handle;
    string command;

    int FirstPlayerWinCount = 0;
    int SencondPlayerWinCount = 0;

    int ex1 = 0;
    int ex2 = 0;
    int ex3 = 0;

    int GamesToPlay = GamesCount;
    bool myFirstMove = MyFirstMove;
    Player myplayer = MyColor;

    Player OpponentPlayer = (myplayer == Player.B ? Player.W : Player.B);
    Player FirstPlayer = (myFirstMove ? myplayer :OpponentPlayer);
    Player SecondPlayer = (!myFirstMove ? myplayer : OpponentPlayer);
    string strFirstPlayer = (FirstPlayer == Player.B ? "Black" : "White");
```csharp
string strSecondPlayer = (FirstPlayer == Player.B ? "White" : "Black");

Tree t;

for (int i = 0; i < GamesToPlay; i++)
{
    HANDLE = CreateFile(pipeName, GENERIC_READ | GENERIC_WRITE, 0, IntPtr.Zero, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, IntPtr.Zero);
    if (handle.IsInvalid)
    {
        ex1++; Thread.Sleep(200);
        handle = CreateFile(pipeName, GENERIC_READ | GENERIC_WRITE, 0, IntPtr.Zero, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, IntPtr.Zero);
    }
    stream = new FileStream(handle, FileAccess.ReadWrite, BUFFER_SIZE, true);
    command = "setupboard"; Console.WriteLine("[Client] Pipe connection established");
    sendMessage(command, stream);
    string line = readmessage(stream);
    Console.WriteLine("{0}: {1}", DateTime.Now, line);
    string[] stones = line.Split(' '); ewn = new EWNBoard(stones[0], stones[2]);
    ewn.PrintBoard();

    stream.Close(); handle.Close();
    startgame = DateTime.Now;
    t = new Tree();
    while (!ewn.EndGame)
    {
        HANDLE = CreateFile(pipeName, GENERIC_READ | GENERIC_WRITE, 0, IntPtr.Zero, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, IntPtr.Zero);
        if (handle.IsInvalid)
        {
            ex2++; Thread.Sleep(200);
            handle = CreateFile(pipeName, GENERIC_READ | GENERIC_WRITE, 0, IntPtr.Zero, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, IntPtr.Zero);
        }
    }
```
stream = new FileStream(handle, FileAccess.ReadWrite, BUFFER_SIZE, true);

////////////////////////////////////////////////////////////////////////
int die = Dice.ThrowDice;
Console.WriteLine(strFirstPlayer + " Dice = " + die + " " + DateTime.Now.ToString());
startmove = DateTime.Now;

//---------The below is player dependent---------
---/---------/
if (myFirstMove)
{
    //Move move = AI.GetMiniMaxMove(ewn, Player.B, die, 0);
    //Move move = AI.GetMiniMaxAlphaBetaMove(ewn, Player.B, die, 0,-1000,1000);
    //Move move = AI.GetMCMove(ewn, myplayer, die, depth);
    Move move = GetMove(Algorithm, ewn, myplayer,die, depth, t);
    endmove = DateTime.Now;
    //Console.WriteLine("MoveTime = " + (endmove - startmove).ToString() + " Rank = " + move.Rank);
    string movetosend = movetoboard(move, ewn);
    ewn.Move(move);
    ewn.PrintBoard();
    sendmessage("play " + myplayer + " " + movetosend, stream);
}
else//OneStone move
{
    sendmessage("genmove " + OpponentPlayer + " " + die, stream);
    line = readmessage(stream);
    Console.WriteLine("{0}: {1}", DateTime.Now, line);
    endmove = DateTime.Now;
    //Console.WriteLine("MoveTime = " + (endmove - startmove).ToString());
    ewn.Move(boardtomove(line.Trim()));
    ewn.PrintBoard();
}

//---------------------------///////////

//////////////////////////////////////////////////////////////
stream.Close();
handle.Close();

if (ewn.Winner == FirstPlayer)
{
    Console.WriteLine(strFirstPlayer + " is Winner");
}
FirstPlayerWinCount++;
}
else
{
    //
    ///////////////////////////////////////////////////////////////////////////////////////////
    handle = CreateFile(pipeName, GENERIC_READ | GENERIC_WRITE, 0, IntPtr.Zero, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, IntPtr.Zero);
    if (handle.IsInvalid)
    {
        ex3++;
        Thread.Sleep(200);
        handle = CreateFile(pipeName, GENERIC_READ | GENERIC_WRITE, 0, IntPtr.Zero, OPEN_EXISTING, FILE_FLAG_OVERLAPPED, IntPtr.Zero);
    }
    stream = new FileStream(handle, FileAccess.ReadWrite, BUFFER_SIZE, true);
    //
    ///////////////////////////////////////////////////////////////////////////////////////////
    die = Dice.ThrowDice;
    Console.WriteLine(strSecondPlayer + " Dice = " + die + " 
" + DateTime.Now.ToString());
    startmove = DateTime.Now;
    //------The below is player dependent------
    if (myFirstMove)
    {
        sendmessage("genmove " + OpponentPlayer + " " + die + " \0", stream);
        line = readmessage(stream);
        Console.WriteLine("{0}: {1}", DateTime.Now, line);
        endmove = DateTime.Now;
        //Console.WriteLine("MoveTime = " + (endmove - startmove).ToString());
        ewn.Move(boardtomove(line.Trim()));
        ewn.PrintBoard();
    }
    else
    {
        //Move move = AI.GetMCMove(ewn, myplayer, die, depth);
        Move move = GetMove(Algorithm, ewn, myplayer, die, depth, t);
        endmove = DateTime.Now;
        //Console.WriteLine("MoveTime = " + (endmove - startmove).ToString() + " Rank = " + move.Rank);
        string movetosend = movetoboard(move, ewn);
        ewn.Move(move);
        ewn.PrintBoard();
        sendmessage("play " + myplayer + " " + movetosend, stream);
    }
}
if (ewn.Winner == SecondPlayer)
{
    Console.WriteLine(strSecondPlayer + " is Winner");
    SencondPlayerWinCount++;
}

stream.Close();
handle.Close();

endgame = DateTime.Now;
Console.WriteLine("GameTime = " + (endgame - startgame).ToString());
Console.WriteLine("End of Game " + i);

Console.WriteLine(myFirstMove ? "My First Move " : "One Stone First Move ");
Console.WriteLine(myFirstMove ? "My Player = " + strFirstPlayer + " One Stone = " + strSecondPlayer : "My Player = " + strFirstPlayer + " One Stone = " + strSecondPlayer);
Console.WriteLine(strFirstPlayer + " =" + FirstPlayerWinCount + " One Stone = " + strSecondPlayer + " = " + SencondPlayerWinCount);

static Move GetMove(string Algorithm, EWNBoard ewn, Player myplayer, int die, int depth, Tree t)
{
    switch (Algorithm.ToLower())
    {
    case "uct":
        Move move = AI.GetUCTMove(ewn, myplayer, die, t);
        return move;
    case "mcts":
        move = AI.GetMCTSMove(ewn, myplayer, die, t);
        return move;
    case "mc":
        move = AI.GetMCMove(ewn, myplayer, die, depth);
        return move;
    case "minimax":
        move = AI.GetMiniMaxMove(ewn, myplayer, die, 0);
        return move;
    case "alphabeta":
        move = AI.GetMiniMaxAlphaBetaMove(ewn, myplayer, die, 0,-1000,1000);
        return move;
    case "random":

move = AI.GetRandomMove(ewn, myplayer, die);
    return move;
default:
    move = AI.GetMCMove(ewn, myplayer, die, depth);
    return move;
}

public static string movetoboard(Move move, EWNBoard board)
{
    string cmdstr = "-";
    {
        cmdstr = "x";
    }
        boardpositions[move.To.X, move.To.Y]);
}

static Move boardtomove(string move)
{
    string[] boardmove = move.Split(new char[] { '-', 'x' });

    Move tempmove = new Move();
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
        {
            if (boardpositions[i, j].Equals(boardmove[0].Trim()))
            {
                tempmove.From = new Space(i, j);
                break;
            }
            if (boardpositions[i, j].Equals(boardmove[1].Trim()))
            {
                tempmove.To = new Space(i, j);
            }
        }
    return tempmove;
}

static void sendmessage(string message, FileStream stream)
{
    ASCIIEncoding encoder = new ASCIIEncoding();
    byte[] messageBuffer = encoder.GetBytes(message);
    stream.Write(messageBuffer, 0, messageBuffer.Length);
    stream.Flush();
}

private static string readmessage(FileStream stream)
{
    byte[] readBuffer = new byte[BUFFER_SIZE];
    ASCIIEncoding encoder = new ASCIIEncoding();
int bytesRead = 0;
bytesRead = stream.Read(readBuffer, 0, BUFFER_SIZE);
string message = encoder.GetString(readBuffer, 0, bytesRead);
return message;
}

private static void readconfiguration(out int GamesCount, out bool MyFirstMove, out Player MyColor, out string Algorithm)
{
try
{
 XmlDocument doc = new XmlDocument();
doc.Load("EWNConfig.xml");

string gamecount;
string firstplayer;
string color;
string algo;

gamecount = doc.SelectSingleNode("//GamesToPlay").InnerText;
firstplayer = doc.SelectSingleNode("//MyFirstMove").InnerText;
color = doc.SelectSingleNode("//MyColor").InnerText;
algo = doc.SelectSingleNode("//Algorithm").InnerText;

GamesCount = int.Parse(gamecount);
MyFirstMove = bool.Parse(firstplayer);
Algorithm = algo;
}
catch (Exception)
{
Console.WriteLine("!!!!!!!!Error Reading Config File. Keeping default settings!!!!!!!!!!!!");
GamesCount = 10;
MyFirstMove = true;
MyColor = Player.B;
Algorithm = "MC";
}
Console.WriteLine("Settings used");
Console.WriteLine("Games to Play = " + GamesCount);
Console.WriteLine("My First Move = " + MyFirstMove);
Console.WriteLine("My Color = " + MyColor);
Console.WriteLine("Algorithm = " + Algorithm);
}
using System;
using System.Collections.Generic;
using System.Text;

namespace EWN
{
    public class Tree
    {
        public Node root;
        public Node pointer;
        public Tree()
        {
            root = new Node();
            pointer = root;
        }

        public Node[] createchildren(Node parent, Move[] m)
        {
            //if (parent.child == null)
            //{
            //    parent.child = new Node[m.Length];
            //    for (int i = 0; i < m.Length; i++)
            //    {
            //        parent.child[i] = new Node(m[i], parent);
            //    }
            //}
            //return parent.child;
            //}

        
        public Node[] createchildren(Node parent, Move[] m, int die)
        {
            if (parent.child[die - 1] == null || parent.child[die - 1].Length == 0)
            {
            }
        }
    }
}
parent.child[die-1] = new Node[m.Length];
for (int i = 0; i < m.Length; i++)
{
    parent.child[die-1][i] = new Node(m[i], parent);
}
return parent.child[die-1];

public class Node
{
    public Node sibling;
    public Node parent;
    public Node[][] child;
    public Move move;
    public Player color;
    public int wins;
    public int visitsnum;

    public Node()
    {
        Move? move = null;
        parent = null;
        wins = -1;
        visitsnum = -1;
        child = new Node[6][];
    }

    public Node(Move m, Node p)
    {
        move = m;
        wins = 0;
        visitsnum = 0;
        parent = p;
        child = new Node[6][];
    }
}