MONTE CARLO TREE SEARCH IN

EINSTEIN WURFELT NICHT!

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

By

Sarmen Shahbazian

May 2012
The graduate project of Sarmen Shahbazian is approved by:

______________________________  ________________________________
Mike Barnes, Ph.D               Date

______________________________  ________________________________
Rick Covington, Ph.D            Date

______________________________  ________________________________
Richard Lorentz, Ph.D, Chair    Date

California State University, Northridge
ACKNOWLEDGMENTS

I would like to thank Richard Lorentz for his suggestion for this project. His help, advice, and motivation made this project possible.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGNATURE PAGE</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vi</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>viii</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 EinStein Würfelt Nicht!</td>
<td>1</td>
</tr>
<tr>
<td>1.2 SteinBot</td>
<td>2</td>
</tr>
<tr>
<td>1.3 SteinBot with Monte Carlo Tree Search</td>
<td>4</td>
</tr>
<tr>
<td>1.4 How Can We Improve MCTS for EWN?</td>
<td>5</td>
</tr>
<tr>
<td>2 MONTE CARLO TREE SEARCH</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Selection</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Expansion</td>
<td>9</td>
</tr>
<tr>
<td>2.4 Simulation</td>
<td>9</td>
</tr>
<tr>
<td>2.5 Backpropagation</td>
<td>10</td>
</tr>
<tr>
<td>3 MONTE CARLO TREE SEARCH IN EINSTEIN WÜRFELT NICHT</td>
<td>11</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Selection</td>
<td>12</td>
</tr>
<tr>
<td>3.3 Expansion</td>
<td>13</td>
</tr>
<tr>
<td>3.4 Simulation</td>
<td>14</td>
</tr>
<tr>
<td>3.5 Backpropagation</td>
<td>15</td>
</tr>
<tr>
<td>4 TEST ENVIRONMENT</td>
<td>16</td>
</tr>
<tr>
<td>4.1 Improving the Simulation Strategy</td>
<td>16</td>
</tr>
<tr>
<td>4.2 Improving the UCT Formula</td>
<td>17</td>
</tr>
<tr>
<td>4.3 Establishing Simulation Timer</td>
<td>17</td>
</tr>
<tr>
<td>4.4 Establishing Number of Simulations before Expanding Children</td>
<td>18</td>
</tr>
<tr>
<td>4.5 Experimenting with Different Settings</td>
<td>18</td>
</tr>
<tr>
<td>4.6 Online Gaming Results</td>
<td>20</td>
</tr>
</tbody>
</table>
4.7 Move Comparisons with other Players ............................................................ 21
4.8 Conclusion .................................................................................................... 24

5 CONCLUSION AND FUTURE RESEARCH ................................................. 25

5.1 Answers to the Research Questions .............................................................. 25
5.2 Future Research .......................................................................................... 25

REFERENCES .................................................................................................. 26

APPENDIX A .................................................................................................... 27

Appendix A.1 SteinBot Class ............................................................................ 27
Appendix A.2 Clone Class ................................................................................ 42
Appendix A.3 Square Class .............................................................................. 54
Appendix A.4 Node Class ................................................................................ 56
LIST OF TABLES

Table 1: Results of Random Simulation against Improved Strategy ................................ 17
Table 2: Results of Improving UCT Formula................................................................. 17
Table 3: Results of 10 Seconds against 60 Seconds ..................................................... 17
Table 4: Establishing number of simulations............................................................... 18
Table 5: Results with 6 or less stones ......................................................................... 19
Table 6: Confirming 6 or less strategy .......................................................................... 19
Table 7: Comparing functions ...................................................................................... 19
Table 8: Results on LittleGolem .................................................................................... 21
Table 9: Move comparison with Sybil_c ........................................................................ 22
Table 10: Move comparison with OneStone_c ............................................................. 23
LIST OF FIGURES

Figure 1: EWN GUI................................................................................................................. 2
Figure 2: Custom Mode GUI ................................................................................................. 3
Figure 3: Outline of MCTS .................................................................................................. 8
Figure 4: Outline of MCTS in EWN .................................................................................. 12
Figure 5: Outline of children for a single node ................................................................. 13
Figure 6: Comparision between SteinBot and Sybil_c ...................................................... 23
ABSTRACT

MONTE CARLO TREE SEARCH IN

EINSTEIN WURFELT NICHT!

By

Sarmen Shahbazian

Master of Science in Computer Science

Monte Carlo simulations have been often used for artificial intelligence with many games. When combined with a search tree and millions of random simulations per second, it can produce a recommended move according to the simulation strategy. But, results can be different for each type of game. The structure and search formula for the tree must be tailored differently for each type of game. In this project, I have created a program called SteinBot for playing Einstein Wurfelt Nicht, which used an improved version of the MCTS algorithm to produce a recommended move. The MCTS was created in a special way to support die values. SteinBot was tested with hundreds of games with the random method and confirmed that improvements are needed for the random simulations and search formula. SteinBot played against many humans with the improved strategy and won 63% of the games.
1 INTRODUCTION

1.1 Einstein Wörfelt Nicht!

Einstein wörfelt nicht! is a board game, designed by Ingo Althöfer, a professor of applied mathematics in Jena, Germany in 2005. Einstein wörfelt nicht! (EWN) has double meanings in German: (1) “When you only have one stone (ein Stein) remaining, you no longer need to play dice, but simply move your stone.” (2) It is a play on Einstein’s famous quote “I am convinced that he (God) does not play dice.” The game is played on a square board with a 5×5 grid (Figure 1). Each player has six stones, numbered one to six. 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, which are placed randomly every time the game starts. The players take turns rolling a six-sided die and then moving the matching stone. If the matching stone is no longer on the board, the player moves a remaining stone whose number is next-highest or next-lowest to the rolled number. The player starting in the top-left may move that stone one square to the right, down, or on the diagonal down and to the right; the player starting in the bottom-right may move that stone one square to the top, left, or on the diagonal up and to the left. Any stone which already lies in the target square is removed from the board. The object of the game is to either get one of your stones to the far corner square in the grid (where your opponent started) or to remove all of your opponent’s stones from the board. [1, 2]
1.2 SteinBot

For my research and experiment, I have created a computer program called SteinBot, which plays EWN with some upgrades and features. It is implemented in Java and using Swing for its Graphical User Interface (GUI). Since it is in Java, object-oriented principles are used and will run on any machine.

SteinBot has three game modes:

- One player (Figure 1): A human player can play against SteinBot with winning results being recorded and will run until the program is closed. The human player is the bottom right blue player which clicks on the stone selected from the dice thrown to be moved (yellow stone) and then clicks on the location to be moved.
• Auto: Two SteinBot’s can play each other with different settings to compare which version produces the most wins for a given number of games played. The settings of the red and blue are configured in the clone class (appendix A.2). This GUI (Figure 1) is the same as one player mode except there will not be any controlled movements by the user.

• Custom (Figure 2): A player can enter the stone locations of each player and the die value, hit the submit button and the recommended best move is displayed for the user (currently for blue player only).

![Custom Mode GUI](image-url)

**Figure 2: Custom Mode GUI**
SteinBot was primarily used in the auto mode for testing the algorithm’s strengths and weaknesses. After running SteinBot on auto mode to play a certain fixed amount of games (e.g. 100), the results were evaluated and recorded to see which player’s settings produced the most wins. Again, the same mode was used with appropriate modifications to the player’s settings and run to produce the winning results. After many tests of different settings, where the win result were impressive, those settings were used with the custom mode to play against another person or “bot” on different game websites, for example, LittleGolem.net. Then, the win results where again evaluated for performance with another player that might be an actual human player or a “bot” with different settings or algorithm. The test results and settings are explained in chapter 4.

1.3 SteinBot with Monte Carlo Tree Search

SteinBot uses Monte Carlo Tree Search (MCTS) for its algorithm. MCTS (see chapter 2 for detailed description) was chosen because it has some advantages over other tree search methods and other games that have used MCTS have proven that it is beneficial. For example, in July 2005, Crazy Stone was developed by Remi Coulom to play one of the first Go programs using the MCTS. In 2006, Crazy Stone took part in its first tournament, winning a gold medal in the 9×9 11th Computer Olympiad. [3]

Some of the benefits for using MCTS with EWN are:

- “It does not require any strategic or tactical knowledge about the given domain to make reasonable decisions. The algorithm can function effectively with no knowledge of a game apart from its legal moves and end conditions.” [4]
• It creates an unbalanced tree that adapts to the search space. The search function selects more interesting nodes to be visited more often and focuses its search time in more relevant parts of the tree. For example, it will pick the branch with the best win ratio from the current node to traverse.

• EWN has a small board size and one die value compared to other board games which could have a board size of 19 × 19 or two die values (e.g. Go and backgammon). With this benefit, the tree will have a reasonable size for memory and faster traversing down the branches.

• The algorithm can be stopped at any time to return the current best node/move. For example, if a child node has a more dominant visit count over the other children, the algorithm can be halted at that point or halted before the time ends and evaluate the board in the remaining time with other strategies.

1.4 How Can We Improve MCTS for EWN?

**Problem Statement:** *How can we improve MCTS to produce the best results for our game?*

MCTS’s appear in many different variants. For example, in 2006 (Coulom, 2006; Kocsis and Szepesvari, 2006; Chaslot et al., 2006a), Coulom added a tree search to Monte Carlo simulations and used his variant to create the first competitive MCTS program, Crazy Stone, which immediately won the 9 × 9 Go tournament at the 11th Computer Olympiad. The variant introduced by Kocsis and Szepesvari (2006), called Upper Confidence Bounds applied to Trees (UCT), was based on the Upper Confidence Bounds (UCB) algorithm (Auer et al., 2002). The variant of Chaslot et al. (2006a) was based on their Objective Monte-Carlo (OMC) algorithm.
Since there have been many additions to Monte Carlo simulations, I have used the version with the tree search and UCT function. This problem statement raises two questions for researching better simulations and UCT parameters.

**Research Question 1:** *How can we improve MCTS random simulations based on our knowledge about the game?*

The MCTS currently uses *random* game simulations for knowledge about the current move. But, since moves are selected from a ratio of wins and visits for a certain move, random simulations will not provide accurate win results. For example, in EWN it is possible to have up to six move options per turn. If a move is randomly picked with the same percentage as the other moves, which could be the best or worst move, the win results will not be accurate. An inaccurate result might cause the wrong move to be picked. So, the random simulations must be enhanced to make moves that are logical to a human player or provide a certain percentage advantage for a certain move. In this research, I have experimented/tested with different options for enhancing the random simulations.

**Research Question 2:** *What parameters in the MCTS produce the best results in our test environment?*

There are many parameters which produce the search results for the tree. For example, the UCT function (see section 2.2), its parameters must be adjusted based on experiments that produced the best results. I propose running many experiments based on previous experiments and comparing results with other programs.
2 MONTE CARLO TREE SEARCH

2.1 Introduction

MCTS is more interesting and useful where building a positional evaluation function can be a difficult time-consuming issue. MCTS consists of two strongly coupled parts: a relatively shallow tree structure and deep simulated games. It uses a best-first search method where it does not require a positional evaluation function. It bases its evaluation on a randomized exploration of the search space. Using the results of previous explorations, the algorithm slowly builds up a game tree in memory, and from millions of simulations becomes better at accurately estimating the values of the most promising moves. It consists of four basic steps, repeated as long as there is time left for the simulation. The steps are the following: (1) In the selection step, the tree is traversed from the root node until it reaches a leaf node $E$, where it selects a position that is not added to the tree yet. (2) Next, during the expansion step, children of $E$ are added to the tree. (3) Then, a random simulated game is played from $E$ until a result is achieved. The result $R$ of this "simulated" game is +1 in case of a win for the “bot” and 0 in case of a win for the opponent. (4) Finally, $R$ is propagated back along the path from $E$ to the root node in the backpropagation step. When time is up, the move picked by the program is the child of the root with the highest value. [5, 6] See the outline of the tree in Figure 3, from [5]. The structure of the tree will be different for each type of game (e.g. Go or EWN), depending on what type of moves are available for a certain turn in a game. Some games involve a die value which decides the appropriate move. In that case the tree must be structured to accommodate all possible moves from the die value (explained in chapter 3).
In this chapter, the generic version will be described and the EWN version will be described in chapter 3.

MCTS has four basic parts: Selection, Expansion, Simulation and Backpropagation.

2.2 Selection

“In the selection phase, the tree is recursively traversed until it reaches a node that is not on the tree yet. It controls the balance between exploitation and exploration. On the one hand, the task often consists of selecting the move that leads to the best results so far (exploitation). On the other hand, the less promising moves still must be tried, due to the uncertainty of the evaluation (exploration).”[8] The objective of the function is to select the next best move from the parent node. The following three are the best strategies used for selection: (1) OMC (Objective Monte-Carlo) by Chaslot (2006). (2) UCT (Upper Confidence bounds applied to Trees) by Kocsis and Szepesvari (2006) (3) PBBM (Probability to be Better than Best Move) by Coulem (2006). The OMC and PBBM functions require too many complicated calculations and implementations, which could result in wasted time. I believe they are still used because newer functions may not yet have proven better win results for their games.
The UCT strategy is the easiest to implement and used in many programs like MoGo [9] which have shown to be impressive against other programs. UCT works as the following. Let I be the set of nodes reachable from the current node p. UCT selects the maximum value child k of the parent node p that satisfies the following formula:

\[ k \in I \left( V_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \right) \]

Equation 1 [6]

Where \( V_i \) is the value (wins / visits) of the node I, \( n_i \) is the visit count of I, and \( n_p \) is the visit count of p. C is a coefficient, which must be determined experimentally.

2.3 Expansion

In the expansion phase, a new node is added to the tree if the selected parent node does not have a child and the predefined limitation is met. The predefined limitation can be: try a set number of random simulations to get a better win ratio for the move from that node or add only one unexplored node to the tree per traversal down that path. The idea of limiting the number of nodes created in the tree is to save memory.

2.4 Simulation

In the simulation (also called a playout) phase, a random game is played from the current node until the end of the game is reached. This simulation may consist of a truly random game or an improved strategy based on knowledge about the game. Pure random (each move has the same probability to be chosen) moves can provide results that are not accurate. For example, if there is a winning move, and another move is selected at random and the opponent wins, that is not accurate because the first player had a winning move. Improving the random simulation has shown improvements in game wins [9]. The simulation can be improved by providing more knowledge based moves rather than just
random. For example, the location of the current move or if a defensive or attack move should be encouraged. The simulation strategy must be implemented in a way so there is a balance between the number of simulations and tree size. If the number of simulations are too large, the tree size will be shallow. If the number of simulations are too small, the tree size will be very large and the strategy will not be evaluated enough for strength. [7]

2.5 Backpropagation
In the backpropagation phase, the result of a simulated game $R$ is traversed backwards using recursion from leaf node $L$ to the nodes it had to traverse to reach $L$ updating the win value of the same player’s nodes visited during that simulation. The value is increased by 1 if the game result is a win and a 0 if the game is loss. The opponent’s nodes will increase by 1 if the current player loses and a 0 if the current player wins.

With regards to the basic steps of the tree search outlined in this chapter, the explanations are used for a general idea of the structure. The structure must be tailored to fit the game it is being used for. For example, some games have a draw result and some do not, the backpropagation results varying. Also, every game has different number of move options, so the nodes in the tree must be created according to the number of move options. As a result, the tree and the simulations must be adjusted according to the game being used. I will discuss the adjustments needed for EWN in chapter 3.
3 MONTE CARLO TREE SEARCH IN EINSTEIN WÜRFELT NICHT

3.1 Introduction
In this chapter, I will discuss how the MCTS has been tailored for EWN. The outline of the tree is shown in Figure 4. I will use Figure 1’s game setup/move to explain the tree structure and the four steps for the tree search in detail in sections 3.2 and 3.3, respectively. In Figure 1, it is Blue’s turn with stone 2 selected to move. Stone 2 has three options (left, diagonal, and up). The tree will be constructed with the root being the moveable stone and the children (in this case, 3 children) being the locations which it can move to. Then, each child created from the blue child represents a move from the red player. The blue child can have up to eighteen children (6 die values × 3 possible moves), each representing a red stone with its moveable options, which can be up to three moveable locations. After a red child is created, then blue children are created from that red child and the tree alternates going down with the parent and child alternating colors (with respect to their moves). In this example, if the game would have started with the red player, the tree structure would be the same, except the root would have started with the red player and its moveable children. Also, there are two conditions where creating the tree is not required: (1) If there is only one location available for the stone to move and (2) if the picked stone has a winning move.
3.2 Selection

Since the UCT formula is easy to implement, does not require complicated calculations and has proven its effectiveness in Go programs [7], it was chosen for EWN. Through my test results, I introduced an extra constant $VK$ to the UCT function.

$$k \in I \left( V_i + C \times \sqrt{\frac{\ln n_p}{VK \times n_i}} \right)$$

Pseudo code of UCTSelect function: (see source code in appendix A.2 Clone class).

```java
n = parent.child
double bestUCT
while (n is not empty)
    UCTValue = (n.wins/n.visits) +
               C*Math.sqrt(Math.log(parent.visits) / (VK* n.visits))
    if(UCTValue > bestUCT)
        bestUCT = UCTValue
        n = n.sibling
    end while
return child with greatest UCTValue
```
3.3 Expansion

During the expansion phase, when a node is selected, the expansion function will check if the current node has a child or not and if the simulation parameter (number of simulations before a child is added to the tree) is met. If it does not have a child and does not meet the simulation condition, it will create the children for the current node in the createChildren function. The number of children created will depend on how many moves are available from that current move. The number can vary from 1 to 6. And since there is a die value used for every move, there are 6 different possible moves from the next turn. So, the children must be created in a unique way. Each node has an array of 6 children for each possible die value. An array was used to store the children because it would be easy to find the child from the die value. The die value minus 1 would be the index for that child. (Note: each node in the array represents a move for the opponent). See Figure 5.

Pseudo code of createChildren function: (see source code in appendix A.2 Clone class).

```plaintext
last = parent
for (every possible location to move) //1 to 6, one for every move
  if (last = parent)
    last.childArray[dieValue-1] = moveable_location_node //first node
  else
    last.sibling = moveable_location_node //2nd to 6th node added
  last = moveable_location_node
end for
```

Figure 5: Outline of children for a single node
3.4 Simulation

As discussed in section 2.4, it is not recommended to play purely random simulations. I experimented with implementing two strategies for better simulations: (1) Increasing the probability of a diagonal move being picked to 75% rather than 33% and (2) making the move smarter, based on strategy knowledge. For example, whenever a diagonal move is available, it is always picked. Since a diagonal move gets the stone closer to the goal, it is encouraged to be the best move. But, only providing a diagonal move, some strategic moves are overlooked. For example, if a move should be defensive. If an opponent is next to the current move and not on the diagonal, it will be overlooked. Another situation would be if the same color stone is next to the move and it should be taken off the board so another stone closer to the goal can move. There are a lot of extra strategies that can be overlooked, but will provide some benefit for the speed of the simulation. Because if a simulation is complicated, the task will be time consuming for the result and fewer simulations will be played in the remaining time. Since implementing a smarter strategy requires knowledge about locations of the other stones and deciding which move has priority, it was very hard for me to program more complicated strategies. Keeping in mind the 2 strategies proposed in this section, the test results are available in chapter 4.

Pseudo code of playSimulation function: (see source code in appendix A.2 Clone class).

```
n = current_node
If (n does not have a child AND visits < number of simulations)
    result = play random game and return which player won
    //0=lose 1=win for current player
else
    next_node = UCTSelect(n) //get the next best child
    result = playSimulation(next_node) //recursion is used
end else
return result
```
3.5 Backpropagation

The result of the simulation is backpropagated to every node it had to traverse to get to that node. The result being a +1 for a win and a 0 for a loss, alternating going back up. For example, if it is the red players turn, and it wins, the current node will be a +1. Then, since the parent is Blue, it will get a result of 0 and again since the parent is red it will get a +1 and so on. The same thing will happen if the current node loses, a result 0 for the current node and +1 for the parent and alternate going back to the root.

Finally, after the predetermined time for the simulation finishes, the child which has the highest visit count from the root is picked as the best move, not the highest win ratio, since the algorithm is concerned about the upper bound confidence of the tree. For example, if a node has a high win ratio, but less visits than the other nodes, it is not considered as the best move because good moves can be explored late in the game and a high win count could just mean it was a lucky move. (All source code is available in Appendix A).
4 TEST ENVIRONMENT

All of the tests were done on an Intel Pentium 4, 3.2 GHz computer, using 2 GB memory. The objective of the testing was to find the best strategy for the simulation and parameters for the UCT function. The testing was first done in auto mode, where two different parameters were selected and evaluated. Then, the best results were used on the popular game forum littlegolem.net to play against other players and algorithms (a.k.a. bots). There are some parameters which need to be described first, they are the following:

- Time = seconds for a simulation to play.
- Visits = number of simulations before a child is added to the tree.
- C = UCT constant.
- VK = constant multiplied to node visits in UCT function.

The total number of games played during a test is 500. When the test starts, the blue player goes first and whoever wins the current game, the loser starts the next game. This way there will not be an advantage with who starts first, it will give both players an opportunity to start a game when lost, because the player which starts first has a small advantage.

4.1 Improving the Simulation Strategy

For the first test, we must understand if using purely random simulations are truly weak. For this test, I proposed two different strategies: (1) increasing the probability of a diagonal move being picked to 75% and (2) always pick a diagonal move when possible. From table 1, we can see that strategy 1 did not help; it lost to pure random by 49%. But,
strategy 2 helped, it won by 52% (261 vs. 239). As a result, strategy 2 will be used for the remaining tests.

**Table 1: Results of Random Simulation against Improved Strategy**

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Time</th>
<th>Visits</th>
<th>C</th>
<th>VK</th>
<th>Wins</th>
<th>Wins %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>10 sec.</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>245</td>
<td>49 %</td>
</tr>
<tr>
<td>2.</td>
<td>10 sec.</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>261</td>
<td>52 %</td>
</tr>
</tbody>
</table>

4.2 Improving the UCT Formula

Now, we must test if adding the proposed VK value to the formula helps. From table 2, we can see that setting the VK value to 10 helps in winning by 52%. As a result, we will keep the value in mind and use it with further testing.

**Table 2: Results of Improving UCT Formula**

<table>
<thead>
<tr>
<th>Time</th>
<th>Visits</th>
<th>C</th>
<th>VK</th>
<th>Wins</th>
<th>Wins %</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 sec.</td>
<td>100</td>
<td>1</td>
<td>10</td>
<td>260</td>
<td>52 %</td>
</tr>
<tr>
<td>10 sec.</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>224</td>
<td>48 %</td>
</tr>
</tbody>
</table>

4.3 Establishing Simulation Timer

Since the number of simulations per move plays a factor in producing a good result, the number of simulations and nodes must be maximized enough for memory usage and end in a reasonable time. The above tests were done using 10 seconds for each move, which took about 24 hours to complete, playing 500 games. We must compare with another reasonable time to see if results improve. Also, the number of simulations and nodes created must be compared. Keeping in mind that the more simulations the better, but more nodes are created and the timer of making a move will also need to be set at a longer duration. In table 3, we compare 10 seconds against 60 seconds.

**Table 3: Results of 10 Seconds against 60 Seconds**

<table>
<thead>
<tr>
<th>Time</th>
<th>Visits</th>
<th>C</th>
<th>VK</th>
<th>Wins</th>
<th>Wins %</th>
<th>Avg. Simulations per Game</th>
<th>Avg. Nodes per Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 sec.</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>237</td>
<td>47 %</td>
<td>1607956</td>
<td>706800</td>
</tr>
<tr>
<td>60 sec.</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>263</td>
<td>53 %</td>
<td>5938958</td>
<td>1458108</td>
</tr>
</tbody>
</table>
Since 60 seconds is a reasonable time to make a move and produces better winning results than 10 seconds, it will be used with further tests. But, the only disadvantage is that it took about 6 days to produce results in auto mode.

### 4.4 Establishing Number of Simulations before Expanding Children

Since the size of the tree is based on how often the child nodes are created, the number of simulations must be tested. The larger the value of the simulations, the tree will be smaller and more information about the current node will be known. The lower the number, the size of the tree will be larger and less information about the current node will be known. So, it is a trade-off between size and knowledge. From table 4, we can see that 100 simulations produced a better winning percent against 10 and 1 visit counts. As a result, 100 will be recommended. A medium value is more interesting because it provides a small amount of information about the node, yet having a big size tree.

#### Table 4: Establishing number of simulations

<table>
<thead>
<tr>
<th>Time</th>
<th>Visits</th>
<th>C</th>
<th>VK</th>
<th>Wins</th>
<th>Wins %</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 sec.</td>
<td>100</td>
<td>1</td>
<td>10</td>
<td>253</td>
<td>51 %</td>
</tr>
<tr>
<td>60 sec.</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>247</td>
<td>49 %</td>
</tr>
<tr>
<td>60 sec.</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>242</td>
<td>48 %</td>
</tr>
</tbody>
</table>

### 4.5 Experimenting with Different Settings

In the beginning of the game, there are a lot of possible moves to reach a winning position. The size of the tree must be large enough to create all the nodes to each possible move, therefore needing a long timer for the simulations. After half way through the game with having 6 or less stones on the board, the end of the game should be close, resulting in a smaller size tree. So, a different setting with Visit = 1 and Timer = 30 seconds was tried when 6 or less stones were on the board. From table 5, we can see that there was an improvement with this strategy.
By using a different strategy when 6 or less stones are on the board, we can see that it improves the win rate. So, in table 6, we confirm this strategy by playing against the same settings, but with a 10 visit count. Again, verifying a 100 visit count is better. When 6 or less stones are on the board, we use a 1 visit count to maximize the tree size and use with 30 seconds. Usually when there are 6 or less stones on the board, the game is near the end and fewer moves are needed to reach the goal.

Table 6: Confirming 6 or less strategy

<table>
<thead>
<tr>
<th>Time</th>
<th>Visits</th>
<th>C</th>
<th>VK</th>
<th>Wins</th>
<th>Wins %</th>
<th>6 or less stones</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 sec.</td>
<td>100</td>
<td>1</td>
<td>10</td>
<td>253</td>
<td>51 %</td>
<td>Visit=1 Time=30 sec.</td>
</tr>
<tr>
<td>60 sec.</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>247</td>
<td>49 %</td>
<td>Not changed</td>
</tr>
</tbody>
</table>

In the article by Brown [10], he stated that the following UCT formula produces the best results with the addition of the extra 2’s.

\[
k \in \{ V_i + 2C \times \sqrt{\frac{2 \ln n_p}{n_i}} \}
\]

So, it was tested against my settings, and table 7 verifies that the proposed formula is a little weaker.

Table 7: Comparing functions

<table>
<thead>
<tr>
<th>Time</th>
<th>Visits</th>
<th>C</th>
<th>VK</th>
<th>Wins</th>
<th>Wins %</th>
</tr>
</thead>
<tbody>
<tr>
<td>60 sec.</td>
<td>100</td>
<td>1</td>
<td>10</td>
<td>253</td>
<td>51 %</td>
</tr>
<tr>
<td>60 sec.</td>
<td>100</td>
<td>2</td>
<td>1</td>
<td>247</td>
<td>49 %</td>
</tr>
</tbody>
</table>
4.6 Online Gaming Results

From table 4, we established the best settings for SteinBot. The settings of: Visit = 100, VK = 10 and a timer of 60 seconds and with Visit = 1 and a timer of 30 seconds when there are 6 or fewer stones on the board. These settings were used to play as many games as possible allotted for the completion of this project against humans and other algorithms (“bots”) on LittleGolem.net (LG) using the custom mode. First, you must start a game. There are several ways in order to start a game: (1) you can register and wait for a player in the waiting room. (2) You can play against another player already in the waiting room (you do not have an option to choose the player, if there are many available). (3) You can invite a player of your choice to play against, from the player’s list. The way to understand if the player is a human or a “bot”, there is a c at the end of the player’s name or there is a word in the name that identifies itself (e.g. RoRo the Bot or OneStone_c). Also, information about the player can be viewed in its profile page to understand what kind of an algorithm or strategy they are using. The advantage of playing on LG is that you can play anyone from anywhere in the world, therefore having a large variation of players. The disadvantage is that players have a time limit of 240 hours to make a move, resulting in a very long wait time for the opponent to make a move. That is not the case of most “bots”, because they are automated. By being automated, they make a move in the time frame of their algorithm, resulting in a faster completion of a game.

SteinBot played with numerous players on LG and gathered data. I have separated my collected data into two groups. (1) Winning results against the humans and (2) “bots”.

20
Table 8 represents the number of games played and results against the humans and the “bots”.

<table>
<thead>
<tr>
<th>Group</th>
<th>Total Games</th>
<th>Wins</th>
<th>Wins %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Humans</td>
<td>115</td>
<td>72</td>
<td>63 %</td>
</tr>
<tr>
<td>Bots</td>
<td>352</td>
<td>165</td>
<td>47 %</td>
</tr>
</tbody>
</table>

From table 8, we see the win rate against the humans is 63% and 47% against the bots. It is expected for SteinBot to win against the humans. But, more games need to be played for more accurate results. Since humans have 240 hours to make a move and not enough human players are available to play, the total number of games played was low. The opposite can be concluded for the bots. There are more “bot” players and they are automated, providing more games and quicker moves. But, the current strategy is still not strong enough against both groups. The simulation strategy needs to be improved.

4.7 Move Comparisons with other Players

Since LG provides a feature where prior games and moves are stored, it is helpful to look at other smarter player’s games and moves to compare with SteinBot’s moves for the purpose of making improvements to the strategy (a.k.a. reverse engineering). I chose two of the best players from the top ten list (based on win/loss ratio) to compare moves. The move comparisons were achieved by entering the stone positions on the board into SteinBot’s custom mode and comparing if the moves matched or were different. The comparison was divided into five categories: (1) top 1st stone at start of game (e.g. blue stone#1 in Figure 2). (2) 1st stone from left on bottom line at start (e.g. blue stone #2 in Figure 2). (3) Bottom 4 corners. (4) Any moves made from the middle of the game. (5) When 6 or less stones are on the board.
Table 9: Move comparison with Sybil_c

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same</td>
<td>2</td>
<td>1</td>
<td>28</td>
<td>38</td>
<td>32</td>
<td>101</td>
</tr>
<tr>
<td>Different</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>25</td>
</tr>
</tbody>
</table>

From table 9, we can see the evaluation for Sybil_c. I made some observations while recording the results. In category 1, the stone sometimes goes diagonal or left, rather than always left. The same with category 2, it either goes up or diagonal, rather than always up. I believe, moving these stones diagonal is a better strategy, because then the opponent is forced to move its stone away from its defensive position, providing a safe path for the attacking player. In the other categories, I made the observation that stones 1 and 6 are kept in play when other options are available. It tends to remove the stone which is between two other values (e.g. remove stone 5, if there exists a stone 6 and 4) to provide more flexibility for the stones. If there are two stones to be moved, it will move the stone closest to the goal, leaving the other one for defensive purposes, especially if the stone is in the starting corner. For a certain play in category 5, see Figure 6, a die value of 1 is thrown and the Blue player must move the 1 stone. SteinBot recommends the 1 stone be moved up but Sybil_c moves diagonal. Myself as a human player, I would move the 1 stone diagonal. After analyzing these moves, moving the 1 stone diagonal is the best move. If the 1 stone is moved diagonal, and the red player also rolls a 1, he will be forced to move right. SteinBot wants to move up so if the red 5 moves right and eliminates the blue 5, the 1 stone will be closer to the goal. Comparing the overall movements with Sybil_c, SteinBot does not try to keep the 1 and 6 stones in play and tries to keep its stones in the middle of the board. Keeping your stones in the middle of the board does not allow an advantage of attacking in the beginning of the game. An attacking strategy is better because it pulls the opponents stones away from their defensive positions.
The next player I chose was OneStone_c. We can see the evaluation in table 10.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same</td>
<td>13</td>
<td>3</td>
<td>36</td>
<td>40</td>
<td>27</td>
<td>119</td>
</tr>
<tr>
<td>Different</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>16</td>
</tr>
</tbody>
</table>

The results were relatively close to SteinBot. The only different observations I made were that it also tends to keep the 1 and 6 stone in play when having a different move option and it tends not to eliminate the opponents stone when another option is also available.

Since OneStone_c is also using MCTS (known from LG profile) for its algorithm, its moves should almost be the same with SteinBot. OneStone_c makes the different moves because of its improved simulation strategy. Sybil_c’s moves are more different because he could be using a different algorithm (his type of algorithm is unknown). It would be difficult to determine which player produces the best moves, because after the move is made, it is up to the opponent to make the anticipated move. For example, if SteinBot
moved his stone diagonal and Sybil_c moved the same stone up, the opponent may move differently and the outcome cannot be predicted. Overall, the movements are almost the same as Sybil_c and OneStone_c.

4.8 Conclusion
From the games played on LG, SteinBot’s algorithm was impressive against humans, but needs more improvements to dominate against the bots. But, in order to have more accurate test results, thousands of games need to be played, because there are some factors which might produce the wrong results (e.g. the setup of the stones on the board). It is a better strategy if the 1 or 6 stone is in the starting corner or on the middle diagonal edge, since the diagonal stone can reach the goal faster and the corner stone can be used for defense, if the opponent approaches its goal. At the beginning of the game, it is better to remove the stones which will help in providing two stone options to be moved. One stone can be used for attacking and the other for defense. Also, since there is a die involved in the game, luck can play a factor. If stone 1 is on the diagonal edge and 1 is rolled three times in a row, the game is won, so strategy is not a factor in this case, producing the wrong result. It was not possible to produce more test results because of time constraints. In auto mode, it took approximately 6 days for 500 games to be played. In custom mode, you must enter the stone locations manually, wait for 60 seconds to produce a move location, and then wait for the opponent to make a move, which can be from 10 seconds to weeks. Although, results show that SteinBot can be a powerful player.
5 CONCLUSION AND FUTURE RESEARCH

5.1 Answers to the Research Questions

Answer to Research Question 1: From table 1, we can see that using the strategy of moving stones diagonal closer to the goal helps significantly than random simulation.

Answer to Research Question 2: After numerous tests, the following parameters produced the best results: Time = 60 seconds, Visits = 100, C = 1.0, VK = 10 and Time = 30 seconds, Visits = 1 when there are 6 or less stones on the board.

5.2 Future Research

For further research, SteinBot can be automated to play online with the other “bots”, without the need to enter the stone positions manually (which is time consuming and the user does not need to be in front of the computer), producing more results sooner. Since SteinBot is created in Java, it is slower compared to C++. From table 3, we can see that SteinBot played 1,607,956 simulations in 10 seconds. When comparing this amount with “RoRoRo the Bot”\(^1\) on LG, we see that he uses C++ and produces 1 million simulation per second, which is about 10 times faster than SteinBot. By having a faster program, more simulations can be played in the allocated time and produce a faster and better move for the waiting opponent. Since SteinBot’s simulation strategy is not very smart, it can be improved and tested to see if better win rates apply. Finally, the UCT function can be tested with different values (e.g. C), to maximize the win rate.

\(^1\)http://fatphil.org/rororo/ai/einstein.html
REFERENCES

1. Wikipedia-Einstein wurfeltnicht
   http://en.wikipedia.org/wiki/EinStein_w%C3%BCrfelt_nicht!

2. BoardGameGeek
   http://www.boardgamegeek.com/boardgame/18699/einstein-wurfelt-nicht

3. Crazy Stone
   http://en.wikipedia.org/wiki/Crazy_Stone_%28software%29

4. Everything Monte Carlo Tree Search
   http://www.mcts.ai/?q=mcts


   http://www.ru.is/faculty/yngvi/pdf/WinandsBS08.pdf


APPENDIX A

Appendix A.1 SteinBot Class

/*
* This is the main SteinBot class
* Creates GUI (each square on GUI is a Square class)
* Has nested Mouse Listener Class
* Must be used with Clone, Square, Node classes (java files)
*/
import java.awt.*;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.swing.*;
import java.awt.event.*;
import java.util.LinkedList;
import javax.swing.text.BadLocationException;

public class SteinBot extends JPanel implements ActionListener {

    JFrame frame; //Swing GUI frame
    private final static int rows = 5; //5 rows on board
    private final static int cols = 5; //5 columns on board
    private int diceVal; //current dice value
    private Square[][] Board; //2D array of board
    //array to know if red stones are on board (index 0 will always be empty)
    private int[] red = new int[7];
    //array to know if blue stones are on board (index 0 will always be empty)
    private int[] blue = new int[7];

    private JButton submit = new JButton("Submit");

    private JLabel infoLabel = new JLabel(" ");
    private JLabel redLabel = new JLabel("Red wins: 0");
    private JLabel blueLabel = new JLabel("Blue wins: 0");
    private JLabel hint = new JLabel(");
    private JLabel lbldice = new JLabel("Dice:");

    private JTextField redvals = new JTextField("---00--00000000000000", 26);
    private JTextField bluevals = new JTextField("00000000000000--00---", 26);
    private JTextField diceval = new JTextField("", 2);

    private JProgressBar progressBar = new JProgressBar();

    private Square displaydice;//for displaying the dice image
private String winner = "";
private String hintString = "";
private String playerTurn = "red"; //current player color

private int redwinner = 0; //red wins
private int bluewinner = 0; //blue wins

//SET GAME MODE HERE//////////

//set game mode to auto, custom, player1
private String gameMode = "custom";

public SteinBot() {
    frame = new JFrame("SteinBot");

    Board = new Square[rows][cols];
    this.setLayout(null);

    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++) {
            Board[i][j] = new Square(i, j, 0, "");
        }

    frame.setContentPane(this);

    infoLabel.setSize(new Dimension(350, 20));
    infoLabel.setLocation(100, 320);
    frame.add(infoLabel);

    redLabel.setSize(new Dimension(350, 20));
    redLabel.setLocation(5, 340);
    frame.add(redLabel);

    blueLabel.setSize(new Dimension(350, 20));
    blueLabel.setLocation(5, 360);
    frame.add(blueLabel);

    hint.setSize(new Dimension(350, 20));
    hint.setLocation(200, 340);
    frame.add(hint);

    submit.setSize(new Dimension(100, 30));
    submit.setLocation(150, 460);
    submit.addActionListener(this);

    redvals.setSize(new Dimension(200, 30));
    redvals.setLocation(100, 380);

    bluevals.setSize(new Dimension(200, 30));
    bluevals.setLocation(100, 420);
diceval.setSize(new Dimension(20, 30));
diceval.setLocation(100, 460);

lblred.setSize(new Dimension(40, 30));
lblred.setLocation(70, 380);

lblblue.setSize(new Dimension(40, 30));
lblblue.setLocation(70, 420);

lbldice.setSize(new Dimension(40, 30));
lbldice.setLocation(70, 460);

displaydice = new Square(0, 0, 0, "dice");
displaydice.setSize(25, 25);
displaydice.setLocation(125, 340);
this.add(displaydice);

progressBar.setStringPainted(true);
progressBar.setIndeterminate(true);
progressBar.setBorderPainted(true);
progressBar.setSize(100, 20);
progressBar.setLocation(200, 340);

progressBar.setString("I'm Thinking");

if (gameMode.equals("custom")) {
    frame.add(submit);
    frame.add(redvals);
    frame.add(bluevals);
    frame.add(diceval);
    frame.add(lblred);
    frame.add(lblblue);
    frame.add(lbldice);
    redLabel.setVisible(false);
    blueLabel.setVisible(false);

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            Square sq = Board[i][j];
            sq.setSize(60, 60);
            sq.setLocation(20 + j * 60, 20 + i * 60);
            this.add(sq);  //add to GUI
        }
    }
} //end custom
if (gameMode.equals("player1")) {
    startGame();
} //end player1

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.addMouseListener(new MyMouseListener());
frame.setSize(350, 600);
frame.setLocation(500, 0);
frame.setVisible(true);

if (gameMode.equals("auto")) {
    //number of games to be played by auto two players
    for (int i = 0; i < 500; i++) {
        startGame();
        numSimsPerGame = 0;
        numNodesPerGame = 0;
        numMoves = 0;
        if (winner.equals("red")) {
            redwinner++; //red is a winner
        } else {
            bluewinner++; //blue is a winner
        }
    } //end for
} //end auto
} //end constructor

//starts game
private void startGame() {
    int x = 1, redp = 0, bluep = 0;
    String player = "red"; //current player turn

    //empty the arrays, set to 0
    for (int i = 0; i < 7; i++) {
        red[i] = 0;
        blue[i] = 0;
    }

    //randomly places stones
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            //if current square needs to be empty
            if (i == 0 && j >= 3 || i == 1 && j >= 2 || i == 2 && (j >= 1 && j != 4)) {
                x = 0;
            }
            //if current square needs to be blue color
            if (i == 2 && j == 4) {
                player = "blue";
            }
        }
    }
//if current square needs to be empty
    if (i == 4 && j <= 1 || i == 3 && j <= 2 || i == 2 && (j <= 3 && j != 0)) {
        x = 0;
    }

if (x != 0) { //if square should not be empty
    if (player.equals("red")) {
        do {
            redp = rollDie(); //find random value
        } while (red[redp] == redp); //if already on board, try again
        Square sq = Board[i][j];
        sq.row = i;
        sq.col = j;
        sq.id = redp;
        sq.playerColor = player;
        sq.setSize(60, 60);
        sq.setLocation(20 + j * 60, 20 + i * 60);
        this.add(sq); //add to GUI
        red[redp] = redp; //record it is on the board
    } //end if(player.equals("red"))

    if (player.equals("blue")) {
        do {
            bluep = rollDie(); //find random value
        } while (blue[bluep] == bluep); //if already on board, try again
        Square sq = Board[i][j];
        sq.row = i;
        sq.col = j;
        sq.id = bluep;
        sq.playerColor = player;
        sq.setSize(60, 60);
        sq.setLocation(20 + j * 60, 20 + i * 60);
        this.add(sq); //add to GUI
        blue[bluep] = bluep; //record it is on board
    } //end if(player.equals("blue"))

    x = 1; //change to 0 if it should not be on board
} else { //empty squares
    Square sq = Board[i][j];
    sq.row = i;
    sq.col = j;
    sq.id = x;
    sq.playerColor = player;
sq.setSize(60, 60);
sq.setLocation(20 + j * 60, 20 + i * 60);
this.add(sq);
x = 1; //change to 0 if it should not be on board
}
} //end 2nd for
} //end 1st for

findStoneToMove(); //find next move

//check if there is a winner
private void checkIfWinner() {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            Board[i][j].repaint();
        }
    }
    boolean flag = false;
    for (int i = 1; i < 7; i++) {
        if (red[i] != 0) { //are there red stones on the board
            flag = false; //yes
            break;
        } else {
            flag = true; //no
            winner = "blue";
        }
    }
    if (flag != true) {
        for (int i = 1; i < 7; i++) {
            if (blue[i] != 0) { //are there blue stones on the board
                flag = false; //yes
                break;
            } else {
                flag = true; //no
                winner = "red";
            }
        }
    }
    //did blue reach top left corner
    if (Board[0][0].id != 0 && Board[0][0].playerColor.equals("blue")) {
        flag = true;
        winner = "blue";
    }
    //did red reach low right corner
    if (Board[4][4].id != 0 && Board[4][4].playerColor.equals("red")) {
        flag = true;
        winner = "red";
    }
if (flag == true) {
    System.out.println(winner + " is a winner");
}

if (!flag) //if there is a winner restart game
{
    findStoneToMove(); //if not find next move
} else if (gameMode.equals("player1")) {

    if (winner.equals("red")) {
        redwinner++; //red is a winner
    } else if (winner.equals("blue")) {
        bluewinner++; //blue is a winner
    }
    redLabel.setText("Red wins: " + Integer.toString(redwinner));
    blueLabel.setText("Blue wins: " + Integer.toString(bluewinner));

    startGame(); //restart game
}

//sets current moveable stone to be moved
private Square setMoveable(int index) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (Board[i][j].id == index && Board[i][j].playerColor.equals(playerTurn)) {
                Board[i][j].moveable = true; //sets moveable square to yellow
                return Board[i][j];
            }
        }
    }
    return Board[0][0]; //should not happen
}

//will roll die, switch player turn and find stones to move
private void findStoneToMove() {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            Board[i][j].moveable = false; //resets
            Board[i][j].selected = false; //resets
        }
    }
    hint.setText(" ");

    Square move = null;
    final LinkedList nodeList = new LinkedList();
if (gameMode.equals("custom")) //get die value from GUI
{
    try {
        diceVal = Integer.valueOf(diceval.getText(0, 1)).intValue();
    } catch (BadLocationException ex) {
        Logger.getLogger(SteinBot.class.getName()).log(Level.SEVERE, null, ex);
    }
} else {
    diceVal = rollDie();  //get random dice value
}

if (playerTurn.equals("red")) {
    playerTurn = "blue";
    if (blue[diceVal] == 0) { //if picked stone is off the board
        for (int i = diceVal; i < 7; i++) {
            if (blue[i] != 0) { //pick next highest stone
                move = setMoveable(i);
                nodeList.add(move);
                break;
            }
        }
        for (int i = diceVal; i > 0; i--) {
            if (blue[i] != 0) { //pick next lowest stone
                move = setMoveable(i);
                nodeList.add(move);
                break;
            }
        }
    } else {
        move = setMoveable(diceVal);
        nodeList.add(move);
    }
} else {//red player turn
    playerTurn = "red";
    if (red[diceVal] == 0) { //if picked stone is off the board
        for (int i = diceVal; i < 7; i++) {
            if (red[i] != 0) { //pick next highest stone
                move = setMoveable(i);
                nodeList.add(move);
                break;
            }
        }
    }
for (int i = diceVal; i > 0; i--) {
    if (red[i] != 0) { //pick next lowest stone
        move = setMoveable(i);
        nodeList.add(move);
        break;
    }
}
else {
    move = setMoveable(diceVal);
    nodeList.add(move);
}
if (gameMode.equals("custom")) {
    nodeList.clear();
}
} //end else

infoLabel.setText(playerTurn + " rolled a: " + diceVal);
displaydice.id = diceVal;
displaydice.repaint(); //update dice image

for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        Board[i][j].repaint();
    }
}
if (!nodeList.isEmpty()) {
    if (gameMode.equals("auto")) {
        getUCTMove(nodeList); //get next move from UCT algorithm
    } else if (gameMode.equals("custom") || gameMode.equals("player1") &&
        playerTurn.equals("red")) {
        //thread to show progress bar
        final Thread t = new Thread() {
            public void run() {
                frame.add(progressBar);
                submit.setEnabled(false);
                setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
                getUCTMove(nodeList); //get next move from UCT algorithm
                submit.setEnabled(true);
                setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
                remove(progressBar);
            }
        };
        t.start();
    }
} //end if
if (gameMode.equals("player1") && playerTurn.equals("red")) {
    checkIfWinner();
}
repaint();
}
};//end else
};//end nodisempty
if (gameMode.equals("auto")) {
    checkIfWinner();
}
};//end findStoneToMove
//roll die
private int rollDie() {
    int rand = (int) (Math.random() * 10000) % 6;
    rand++;
    return rand;//return 1-6
}
//Event listiner for buttons
public void actionPerformed(ActionEvent e) {
    //submit btn, used with custom board
    if (e.getSource() == submit) {
        int loc = 0;
        String player = ";"; //player turn
        //empty the arrays, set to 0
        for (int i = 0; i < 7; i++) {
            red[i] = 0;
            blue[i] = 0;
        }
        //custom placing stones from GUI inputs
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                try {
                    int tempnum = Integer.valueOf(redvals.getText(loc, 1)).intValue();
                    red[tempnum] = tempnum;
                }
            }
        }
    }
}
player = "red";

if (tempnum == 0) {
    tempnum = Integer.valueOf(bluevals.getText(loc, 1)).intValue();
    blue[tempnum] = tempnum;
    player = "blue";
}

Square sq = Board[i][j];
sq.row = i;
sq.col = j;
sq.id = tempnum;
sq.playerColor = player;

sq.setSize(60, 60);
sq.setLocation(20 + j * 60, 20 + i * 60);
this.add(sq);  //add to GUI

} catch (BadLocationException ex) {
    Logger.getLogger(SteinBot.class.getName()).log(Level.SEVERE, null, ex);
}
loc++;
}  //end 2nd for
}  //end 1st for

findStoneToMove();  //find next move

}  //end submit btn

}  //end actionPerformed

//displays hint string
private void displayHint() {
    hint.setText(hintString);
}

//Nested mouse listener class to listen for mouse clicks
class MyMouseListener extends JPanel implements MouseListener {

    public void mousePressed(MouseEvent e) {
        int x = e.getX();  //mouse clicked location
        int y = e.getY();

        int col = (x + 35) / 60;
        int row = (y + 10) / 60;
        col--;
        row--;
//if clicked on board
if (col > -1 && col < 5 && row > -1 && row < 5) {
    if (Board[row][col].moveable == true) {
        Board[row][col].selected = true; //when clicked on moveable stone
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                if (i != row && col != j) {
                    Board[i][j].selected = false; //reset others
                }
            }
        }
    }
}

//checks if blue can move left
if (playerTurn.equals("blue") && col < 4) {
    if (Board[row][col + 1].selected == true) {
        int tempid = Board[row][col].id; //get target square id
        String tempcolor = Board[row][col].playerColor; //current stone color
        Board[row][col].id = Board[row][col + 1].id; //change target id
        Board[row][col + 1].id = 0; //change from square id to 0
        Board[row][col].playerColor = "blue"; //target stone color
        if (tempid != 0) { //if there is a stone on target square
            if (tempcolor.equals("red")) {
                red[tempid] = 0; //remove from board
            }
            if (tempcolor.equals("blue")) {
                blue[tempid] = 0; //remove from board
            }
        }
        checkIfWinner(); //check if there is a winner
    }
}

//checks if blue can move diagonal
if (playerTurn.equals("blue") && col < 4 && row < 4) {
    if (Board[row + 1][col + 1].selected == true) {
        int tempid = Board[row][col].id;
        String tempcolor = Board[row][col].playerColor;
        Board[row][col].id = Board[row + 1][col + 1].id;
        Board[row + 1][col + 1].id = 0;
        Board[row][col].playerColor = "blue";
        if (tempid != 0) {
            if (tempcolor.equals("red")) {
                red[tempid] = 0;
            }
            if (tempcolor.equals("blue")) {
                blue[tempid] = 0;
            }
        }
        checkIfWinner();
    }
}
//checks if blue can move up
if (playerTurn.equals("blue") && row < 4) {
    if (Board[row + 1][col].selected == true) {
        int tempid = Board[row][col].id;
        String tempcolor = Board[row][col].playerColor;
        Board[row][col].id = Board[row + 1][col].id;
        Board[row + 1][col].id = 0;
        Board[row][col].playerColor = "blue";
        if (tempid != 0) {
            if (tempcolor.equals("red")) {
                red[tempid] = 0;
            }
            if (tempcolor.equals("blue")) {
                blue[tempid] = 0;
            }
        }
        checkIfWinner();
    }
}

//checks if red can move right
if (playerTurn.equals("red") && col > 0) {
    if (Board[row][col - 1].selected == true) {
        int tempid = Board[row][col].id;
        String tempcolor = Board[row][col].playerColor;
        Board[row][col].id = Board[row][col - 1].id;
        Board[row][col - 1].id = 0;
        Board[row][col].playerColor = "red";
        if (tempid != 0) {
            if (tempcolor.equals("red")) {
                red[tempid] = 0;
            }
            if (tempcolor.equals("blue")) {
                blue[tempid] = 0;
            }
        }
        checkIfWinner();
    }
}

//checks if red can move down
if (playerTurn.equals("red") && row > 0) {
    if (Board[row - 1][col].selected == true) {
        int tempid = Board[row][col].id;
        String tempcolor = Board[row][col].playerColor;
        Board[row][col].id = Board[row - 1][col].id;
        Board[row - 1][col].id = 0;
        Board[row][col].playerColor = "red";
        if (tempid != 0) {
            if (tempcolor.equals("red")) {
                red[tempid] = 0;
            }
if (tempcolor.equals("blue")) {
    blue[tempid] = 0;
}
}
checkIfWinner();
}
//checks if red can move diagonal
if (playerTurn.equals("red") && row > 0 && col > 0) {
    if (Board[row - 1][col - 1].selected == true) {
        int tempid = Board[row][col].id;
        String tempcolor = Board[row][col].playerColor;
        Board[row][col].id = Board[row - 1][col - 1].id;
        Board[row - 1][col - 1].id = 0;
        Board[row][col].playerColor = "red";
        if (tempid != 0) {
            if (tempcolor.equals("red")) {
                red[tempid] = 0;
            }
            if (tempcolor.equals("blue")) {
                blue[tempid] = 0;
            }
        }
    }
    checkIfWinner();
}
}
//end if(col < 5 && row < 5)

//end of mousepressed

public void mouseReleased(MouseEvent e) {
    //System.out.println("released at "+e.getX()+","+e.getY());
}

public void mouseClicked(MouseEvent e) {
    //System.out.println("clicked at "+e.getX()+","+e.getY());
}

public void mouseEntered(MouseEvent e) {
    //System.out.println("entered at "+e.getX()+","+e.getY());
}

public void mouseExited(MouseEvent e) {
    //System.out.println("exited at "+e.getX()+","+e.getY());
}
//get the best move from UCT in clone class
private void getUCTMove(LinkedList n) {
    Clone clone = new Clone(); //create clone class

    Node temp = clone.UCTSearch(n, Board, red, blue);

    if (playerTurn.equals("blue")) {
        numSimsPerGame += clone.numSims;
        numNodesPerGame += clone.numNodes;
        numMoves++;
    }

    if (gameMode.equals("custom")) {
        //hint for blue player and in custom mode
        if ((temp.fromX - temp.col) == 1)//left
            hintString = "Move stone " + temp.id + " left";
        }

        if ((temp.fromY - temp.row) == 1)//up
            hintString = "Move stone " + temp.id + " up";
        }

        if ((temp.fromY - temp.row) == 1 && (temp.fromX - temp.col) == 1)//diagonal
            hintString = "Move stone " + temp.id + " diagonal";
        }

        displayHint();
    } else if (gameMode.equals("auto")
        || gameMode.equals("player1") && temp.playerColor.equals("red")) {
        makeMove(temp);
    }
}

//move the choosen node to target square
private void makeMove(Node n) {

    int tempid = Board[n.row][n.col].id;
    String tempcolor = Board[n.row][n.col].playerColor;
    Board[n.row][n.col].id = Board[n.fromY][n.fromX].id;
    Board[n.fromY][n.fromX].id = 0;
    Board[n.row][n.col].playerColor = Board[n.fromY][n.fromX].playerColor;
}
if (tempid != 0) {                //if there is a stone on target square
    if (tempcolor.equals("red")) //remove stone from target square
    {
        red[tempid] = 0;           //remove red
    }
    if (tempcolor.equals("blue")) {
        blue[tempid] = 0;          //remove blue
    }
}

public static void main(String args[]) {
    new SteinBot();
}

Appendix A.2 Clone Class

/*
 *This class provides the functions for creating the Monte Carlo tree and
 *playing the simulations
 *
 * Tree search functions have been researched from http://senseis.xmp.net/?UCT%2FDiscussion
 */

import java.util.LinkedList;

public class Clone {

    private int rows = 5, cols = 5;
    private int diceVal;                               //current dice value
    private Node[][] cloneBoard = new Node[rows][cols]; //init board
    //array to know if red stones are on board (index 0 will always be empty)
    private int[] clonered = new int[7];
    //array to know if blue stones are on board (index 0 will always be empty)
    private int[] cloneblue = new int[7];
    private String playerTurn = "red";     //current player turn color
    private String winner = "";            //which color is winner
    private String mainPlayerTurn = "";
    private double localvisits;
    private double UCTK;   //UCT constant
    private int VK;        //used in UCT function
    public int numSims = 0;
    public int numNodes = 0;
    private Node root;   //tree root

    public Clone() {
        startGame();   //init clone board
    }
private void startGame() {
    for (int i = 0; i < 7; i++) {
        clonered[i] = 0;
        cloneblue[i] = 0;
    }
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cloneBoard[i][j] = new Node(i, j, 0, " ");
        }
    }
}

private boolean isGameOver() {
    boolean flag = false;
    for (int i = 1; i < 7; i++) {
        if (clonered[i] != 0) { //is there a red stone on the board
            flag = false;
            break;
        } else {
            flag = true;
            winner = "blue";
        }
    }
    if (flag != true) { //if there is a winner, skip for (int i = 1; i < 7; i++) {
        if (cloneblue[i] != 0) { //is there a blue stone on the board
            flag = false;
            break;
        } else {
            flag = true;
            winner = "red";
        }
    }
    if (flag != true) { //if there is a winner, skip
        if (cloneBoard[0][0].id != 0 && cloneBoard[0][0].playerColor.equals("blue")) {
            flag = true;
            winner = "blue";
        }
    }
    if (cloneBoard[4][4].id != 0 && cloneBoard[4][4].playerColor.equals("red")) {
        flag = true;
        winner = "red";
    }
}
return flag;
}

//find location on board for picked move
private Node findLocationOnBoard(int index) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (cloneBoard[i][j].id == index && cloneBoard[i][j].playerColor.equals(playerTurn)) {
                return cloneBoard[i][j];
            }
        }
    }
    return cloneBoard[0][0];    //should never happen
}

//will roll die, switch player turn and find stones to move
private LinkedList findStoneToMove() {
    LinkedList nodeList = new LinkedList();

    Node move = null;

diceVal = rollDie();    //roll die
if (playerTurn.equals("red")) {
    playerTurn = "blue";
    if (cloneblue[diceVal] == 0) {              //if the stone is not on board
        for (int i = diceVal; i < 7; i++) //find next highest stone
        {
            if (cloneblue[i] != 0) {
                move = findLocationOnBoard(i);
                nodeList.add(move);
                break;
            }
        }
    for (int i = diceVal; i > 0; i--) //find next lowest stone
    {
        if (cloneblue[i] != 0) {
            move = findLocationOnBoard(i);
            nodeList.add(move);
            break;
        }
    }
} else {
    move = findLocationOnBoard(diceVal);
    nodeList.add(move);
}
} else {
    playerTurn = "red";
    if (clonered[diceVal] == 0) {       //if the stone is not on board
for (int i = diceVal; i < 7; i++) //find next highest stone
{
    if (clonered[i] != 0) {
        move = findLocationOnBoard(i);
        nodeList.add(move);
        break;
    }
}
for (int i = diceVal; i > 0; i--) {
    if (clonered[i] != 0) { //find next lowest stone
        move = findLocationOnBoard(i);
        nodeList.add(move);
        break;
    }
} else {
    move = findLocationOnBoard(diceVal);
    nodeList.add(move);
}
return nodeList;

//return random die value
private int rollDie() {
    int rand = (int) (Math.random() * 10000) % 6;
    rand++;
    return rand;//return 1-6
}

// generate a move, using the uct algorithm
public Node UCTSearch(LinkedList moveableNodes, Square Board[][], int red[], int blue[]) {
    LinkedList nodeList = new LinkedList();
    int val = 0;
    for (int i = 0; i < moveableNodes.size(); i++) //loop if more then one stone can move
    {
        Square moveable = (Square) moveableNodes.get(i);
        //convert Square list to Node
        Node currentmove = new Node(moveable.row, moveable.col, moveable.id, moveable.playerColor);
        mainPlayerTurn = moveable.playerColor;
        nodeList.add(currentmove);
        if (val == 0) {
            val = moveable.id;
        }
    }
}
int time = 5000;
int localcount = 0;
if (mainPlayerTurn.equals("blue")) {
    localvisits = 10;
    UCTK = 1.0;
    VK = 10;
    // time = 10000;
    for (int i = 0; i < red.length; i++) {
        if (red[i] != 0) {
            localcount++;
        }
    }
    for (int i = 0; i < blue.length; i++) {
        if (blue[i] != 0) {
            localcount++;
        }
    }
    if (localcount < 7) {
        localvisits = 1;
        //UCTK = 0.6;
        //VK = 1;
        time = 30000;
    }
} else {
    localvisits = 1;
    UCTK = 1.0;
    VK = 1;
    //time = 10000;
}

root = new Node(0, 0, val, "root"); //init uct tree

int numOfChildren = createChildren(root, nodeList);

LinkedList currentNodeList = getMoveableLocation(nodeList, false);

//if only one move or winning move, do not play simulations
for (int i = 0; i < currentNodeList.size(); i++) {
    Node moveable = (Node) currentNodeList.get(i);
    if (moveable.row == 0 && moveable.col == 0 || moveable.row == 4 && moveable.col == 4 || numOfChildren == 1) {
        try {
            Thread.sleep(5000);
        }
    }
}
try {
    catch (InterruptedException ex) {
        return moveable;
    }
}

long startTime = System.currentTimeMillis();  // simulation start time
long endTime = System.currentTimeMillis();  // end simulation

while ((endTime - startTime) < time) {  // number of simulation for given seconds
    copyStateFrom(Board, red, blue);
    playSimulation(root);
    numSims++;
    endTime = System.currentTimeMillis();
}

Node n = getBestChild(root);  // get best move
return n;  // return best move to master board

// copy stone locations from original board
private void copyStateFrom(Square Board[][], int red[], int blue[]) {
    for (int x = 0; x < rows; x++) {
        for (int j = 0; j < cols; j++) {
            cloneBoard[x][j].id = Board[x][j].id;
            cloneBoard[x][j].playerColor = Board[x][j].playerColor;
            cloneBoard[x][j].col = j;
            cloneBoard[x][j].row = x;
        }
    }
    cloneblue = blue.clone();
    clonered = red.clone();
}

// get location where chosen stones can move
// creates nodes for tree
private LinkedList getMoveableLocation(LinkedList nodeList, boolean upgradeRandomGame) {
    LinkedList moveableList = new LinkedList();
    // loop for all the move
    for (int s = 0; s < nodeList.size(); s++) {
        Node moveable = (Node) nodeList.get(s);  // check all moveable stones
        if (moveable.playerColor.equals("red")) {  // red movements
            for (int i = 0; i < rows; i++) {
                for (int j = 0; j < cols; j++) {
                    // code here
                }
            }
        }
    }
}
if ((i - 1) == moveable.row && (j - 1) == moveable.col) // going diagonal down
(i - 1) == moveable.row && j == moveable.col || // going down
i == moveable.row && (j - 1) == moveable.col) { // going right

Node node = new Node(i, j, moveable.id, moveable.playerColor);
node.fromX = moveable.col;
node.fromY = moveable.row;
moveableList.add(node);

///// ALL DIAGONAL /////
//// COMMENT OUT IF RANDOM IS REQUIRED /////

if ((i - 1) == moveable.row && (j - 1) == moveable.col &&
    upgradeRandomGame && mainPlayerTurn.equals("red")) {
    node = new Node(i, j, moveable.id, moveable.playerColor);
    node.fromX = moveable.col;
    node.fromY = moveable.row;
    moveableList.clear();

    if (nodeList.size() == 1) {// if only one stone to move
        moveableList.add(node);
        return moveableList;
    } else {
        Node moveable1 = (Node) nodeList.get(0);
        Node moveable2 = (Node) nodeList.get(1);
        // check which is closest to goal
        if (moveable1.col >= moveable2.col && moveable1.row >=
            moveable2.row) {
            moveableList.add(node);
        } else if (moveable2.col >= moveable1.col && moveable2.row >=
            moveable1.row) {
            i = moveable2.row;
            j = moveable2.col;

            if (moveable2.col + 1 != 5) {
                j = moveable2.col + 1;
            }
            if (moveable2.row + 1 != 5) {
                i = moveable2.row + 1;
            }

            node = new Node(i, j, moveable2.id, moveable2.playerColor);
            node.fromX = moveable2.col;
            node.fromY = moveable2.row;
            moveableList.add(node);
        } else {
            moveableList.add(node);
        }
    }
}
return moveableList;
}
}

//////////END DIAGONAL
}
} //end for
}
} //end if
else {

///////////BLUE MOVEMENTS/////////
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < rows; j++) {
        if ((i + 1) == moveable.row && (j + 1) == moveable.col ||  
            (i + 1) == moveable.row && j == moveable.col ||  
            i == moveable.row && (j + 1) == moveable.col) {  //going diagonal up
            Node node = new Node(i, j, moveable.id, moveable.playerColor);
            node.fromX = moveable.col;
            node.fromY = moveable.row;
            moveableList.add(node);
        }
    }
}

Node node = new Node(i, j, moveable.id, moveable.playerColor);
node.fromX = moveable.col;
node.fromY = moveable.row;
moveableList.add(node);

///////////ALL DIAGONAL///////////

int index = nodeList.size();
for (int i = 0; i < index; i++) {
    moveableList.add(node);
}

if (nodeList.size() == 1) {//if only one stone to move
    moveableList.add(node);
    return moveableList;
} else {
    Node moveable1 = (Node) nodeList.get(0);
    Node moveable2 = (Node) nodeList.get(1);
    if (moveable1.col <= moveable2.col && moveable1.row <= 
        moveable2.row) {
        moveableList.add(node);
    } else if (moveable2.col <= moveable1.col && moveable2.row <= 
        moveable1.row) {
        i = moveable2.row;
        j = moveable2.col;

        if (moveable2.col - 1 != -1) {
            j = moveable2.col - 1;
        }
    }
} //if only one stone to move

return moveableList;
}
i = moveable2.row - 1;
}

node = new Node(i, j, moveable2.id, moveable2.playerColor);
node.fromX = moveable2.col;
node.fromY = moveable2.row;
moveableList.add(node);
} else {
moveableList.add(node);
}

return moveableList;
}  //end else

}  //END ALL DIAGONAL///////

}  //end for (int j=0; j<rows; j++)

}  //end blue else

} //end for (int s = 0; s <nodeList.size(); s++)

return moveableList;

}  // expand children in Node
private int createChildren(Node parent, LinkedList nodeList) {
  Node last = parent;

  //nodeList is the squares where moveable stones can move
  nodeList = getMoveableLocation(nodeList, false);

  //creates children with all possible moves
  for (int i = 0; i < nodeList.size(); i++) {
    Node node = (Node) nodeList.get(i);
    if (last == parent) {
      last.nodeArray[node.id - 1] = node;  //create first child
      numNodes++;
    } else {
      last.sibling = node;  //create other children
      numNodes++;
    }
    last = node;
  }  //end for

  return nodeList.size();  //how many children
}
private int playSimulation(Node n) {
    LinkedList nodeList = new LinkedList();
    int val;

    // change player turn color
    if (n.playerColor.equals("red") || n.playerColor.equals("root")) {
        playerTurn = "red";
    } else {
        playerTurn = "blue";
    }

    if (!n.playerColor.equals("root")) { // if n is not the root
        nodeList = findStoneToMove();
        val = ((Node) nodeList.getFirst()).id;
    } else {
        val = n.id;
    }

    int randomresult = 0;
    if (n.nodeArray[val - 1] == null && n.visits < localvisits) {
        randomresult = playRandomGame(nodeList);

        if (n.playerColor.equals("blue") && randomresult == 0) {
            randomresult = 1;
        } else if (n.playerColor.equals("blue") && randomresult == 1) {
            randomresult = 0;
        }
    } else {
        if (n.nodeArray[val - 1] == null) {
            createChildren(n, nodeList);
        }
    }

    Node next = UCTSelect(n, val); // select a move

    makeMove(next); // make move

    int res;
    if (isGameOver()) { // is gameover after moved
        res = getWinner(); // who is the winner
        next.updateVisit(); // update visits
        next.updateWin(1); // update win
        if (n.playerColor.equals("red") && res == 1) {
            res = 0;
        } else if (n.playerColor.equals("red") && res == 0) {

51
```java
res = 1;
}
} else {
    res = playSimulation(next);
}
res = 1 - res;
randomresult = res;
} //end else

n.updateWin(randomresult); //update node win
n.updateVisit();        //update node visit
return randomresult;
}

// return 0=lose 1=win for current player to move
private int playRandomGame(LinkedList locations) {
    LinkedList nodeList = new LinkedList();
    while (true) { //while game is not over
        nodeList = getMoveableLocation(locations, true);
        //randomly choose a move, if one node is returned then not random
        makeMove((Node) nodeList.get((int) (Math.random() * 1000) % nodeList.size()));

        if (isGameOver()) {
            break; //if game is over
        }
        locations = findStoneToMove();
    }
    return getWinner();
}

//who is the current winner
private int getWinner() {
    if (winner.equals("red")) {
        return 1; //red is a winner
    } else {
        return 0; //blue is a winner
    }
}

// child with highest number of visits is used (not: best winrate)
private Node getBestChild(Node root) {
    //Node child = root.child; old
    Node child = root.nodeArray[root.id - 1]; //child of root from array
    ```
Node best_child = null;
int best_visits = -1;

while (child != null) { // for all children
    // System.out.println("nwins "+child.wins);
    // System.out.println("visits "+child.visits);
    if (child.visits > best_visits) {
        best_child = child;
        best_visits = child.visits;
    }
    child = child.sibling;
} //System.out.println("*******");
return best_child;

//get the child with best UCT value
private Node UCTSelect(Node node, int val) {
    Node res = null;
    Node next = node.nodeArray[val - 1];//val=current die value
double best_uct = -1.0;
double uct = 0.0;
    while (next != null) { // for all children
        double uctvalue;
        if (next.visits > 0) {
            double winrate = (double) next.wins / next.visits;
            uct = UCTK * Math.sqrt(Math.log(node.visits) / (VK * next.visits));
            uctvalue = winrate + uct;
        } else {
            // Always play a random unexplored move first
            uctvalue = 10000 + 1000 * Math.random();
        }
        if (uctvalue > best_uct) { // get max uctvalue of all children
            best_uct = uctvalue;
            res = next;
        }
        next = next.sibling;
    }
    return res;
}
//move the choosen node to target square
private void makeMove(Node n) {
    int tempid = cloneBoard[n.row][n.col].id;
    String tempcolor = cloneBoard[n.row][n.col].playerColor;
    cloneBoard[n.row][n.col].id = cloneBoard[n.fromY][n.fromX].id;
    cloneBoard[n.fromY][n.fromX].id = 0;
    cloneBoard[n.row][n.col].playerColor = cloneBoard[n.fromY][n.fromX].playerColor;

    if (tempid != 0) {                //if there is a stone on target square
        if (tempcolor.equals("red")) //remove stone from target square
            { 
            clonered[tempid] = 0;           //remove red
            }
        else if (tempcolor.equals("blue")) {
            cloneblue[tempid] = 0;          //remove blue
        }
    }
}
}//end clone class

Appendix A.3 Square Class

/*
 * Holds information about GUI squares
 * Loads stone images (must have an images file)
 */
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.*;
import java.awt.*;
import javax.swing.*;
class Square extends JPanel {
    public int row, col, id;
    public String playerColor;
    public boolean moveable;
    public boolean selected;
    private Image blueImg[] = new Image[6];
    private Image redImg[] = new Image[6];
    private Image blueYellowImg[] = new Image[6];
    private Image redYellowImg[] = new Image[6];
    private Image dice[] = new Image[6];

    public Square(int r, int c, int x, String y) {
        row = r;        //row coordinate on 2D board array
        col = c;        //column coordinate on 2D board array
        id = x;        //stone number
        playerColor = y; //stone color
moveable = false; // is it moveable
selected = false; // was it clicked on
// loads images, images must be available to load in file
for (int i = 0; i < 6; i++) {
    blueImg[i] = load_picture("bb" + (i + 1) + ".png");
    redImg[i] = load_picture("rb" + (i + 1) + ".png");
    blueYellowImg[i] = load_picture("by" + (i + 1) + ".png");
    redYellowImg[i] = load_picture("ry" + (i + 1) + ".png");
    dice[i] = load_picture("die" + (i + 1) + ".gif");
}

private Image load_picture(String imageFile) {
    BufferedImage image = null;
    String fullname = "../images" + imageFile;
    try {
        image = ImageIO.read(new File(fullname));
    } catch (IOException e) {
        System.out.println("error loading image " + fullname);
    }
    return image;
}

public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);
    int x = 5;
    int y = 5;

    if (id != 0) {
        g.drawString("" + id, (getSize().width - 1) / 2, (getSize().height - 1) / 2);  

        if (playerColor.equals("red")) {
            // g.setColor(Color.red);
            // g.drawRect(10, 10, 25, 25);
            g.drawImage(redImg[id - 1], x, y, null);
            if (moveable == true) {
                g.drawImage(redYellowImg[id - 1], x, y, null);
            }
        }
        if (playerColor.equals("blue")) {
            // g.setColor(Color.blue);
            // g.drawRect(10, 10, 25, 25);
            g.drawImage(blueImg[id - 1], x, y, null);
            if (moveable == true) {
                g.drawImage(blueYellowImg[id - 1], x, y, null);
            }
        }
        // display dice
        if (playerColor.equals("dice")) {
            g.drawImage(dice[id - 1], 0, 0, null);
        }
    }
Appendix A.4 Node Class

/*
 * This class stores information about nodes on the tree
 */

public class Node {

    public int row, col, id;
    public String playerColor;
    public int wins = 0;
    public int visits = 0;
    public int fromX;    //from X location
    public int fromY;    //from y location
    public Node sibling;
    public Node[] nodeArray = new Node[6];    //children

    public Node(int r, int c, int x, String y) {
        row = r;
        col = c;
        id = x;
        playerColor = y;
    }

    public void updateWin(int val) {
        wins += val;
    }

    public void updateVisit() {
        visits++;
    }

}   //end Node class