University of Otago

Monte-Carlo Tree Search in the game of Tantrix

Cosc490 Final Report

Student **Pete Bruns (ID: 3526384)**
Supervisor **Michael Albert**

Abstract

In the complex game of Tantrix current approaches to artificial intelligence fail to beat even moderate human players. Monte Carlo tree search has been proven to be successful in games such as Go, Backgammon, and Poker. This report covers the design, construction and testing of a Tantrix playing program which implements the Monte Carlo tree search strategy. The program was able to defeat the previously best performing Tantrix bots.

Table of Contents

Chapter 1: Introduction

Traditional artificial intelligence (AI) approaches to searching game trees rely heavily on distinguishing between good and bad game scenarios. They firstly create a game tree to represent possible future game scenarios. These trees are often very large and usually can only be partially created or searched. A static evaluation function is applied to score the quality of the leaf nodes, allowing them to be compared. By comparing leaf nodes we can determine the best scenario to target. These scores are then propagated back up the tree to provide information for choosing the best possible move.

The key element of traditional approaches is the static evaluation function. This also applies to the refinements and optimisations on these traditional approaches, such as alpha-beta pruning. For complex games, these evaluation functions are often difficult to create. As traditional AI approaches rely on static evaluation functions they often have limited success with complex games. Conversely, Monte Carlo tree search (MCTS) removes the need for a static evaluation function by replacing it with results from a number of randomly played games. These games are played out from selected leaf nodes, thereby informing us on specific areas of interest within the tree.

Additionally, MCTS incorporates a dynamically growing game tree, trading off exploration of every possible move with an exploitative strategy of more frequent and hence deeper analysis of promising moves. Recently MCTS has shown great promise in the game of Go, offering significant improvements over any previous techniques [4].

One such game that makes finding a suitable evaluation function difficult is the game of Tantrix, due to its constantly changing board shape and complex rules. This report aims to investigate whether Monte Carlo tree search is a viable algorithm for this game. Tantrix was chosen as it is a game of imperfect information, which has elements of chance combined with a set of complex rules. This makes designing a suitable evaluation function difficult, subsequently allowing the existing bot designs to be beaten by moderately skilled human players [1]. For these reasons, I believe Tantrix is an interesting candidate for MCTS, an algorithm that has already been proven to be successful for games such as Go [4] and Backgammon [13].

The Rules of Tantrix

Tantrix is an abstract strategy game invented by New Zealander Mike McManaway in 1988. The earlier versions of the game were called Mind Game and involved hexagonal tiles crafted out of cardboard. This game only had tiles with two differently coloured lines therefore restricting it to only two players. Gradually the rules and tiles have been adjusted to include two more colours into the game and a whole new set of tiles. In 1992 the game of Tantrix became as we know it today.

In Tantrix there are 56 unique hexagonal tiles which share some similarities, as shown in figure 1. Each piece is one of four shapes: sint (single intersection), brid (bridge), chin (Chinese character), and rond (roundabout).

Figure 1: The four types of Tantrix tile (from Tantrix.com [9])

Tantrix is played with a maximum of four players, although this project concentrated on the more strategic two-player version of the game.

At the start of the game each player is assigned his/her own colour (red, yellow, blue or green). Players take turns in placing tiles while aiming to create the longest line or loop of their assigned colour. The score for each player is made by the number of tiles involved in his/her longest line. If a line is closed into a loop each tile involved counts as two points. The game ends when every tile has been played and the player with the most points wins.

A player's hand consists of six tiles. Each time a tile is played a replacement tile is drawn from the bag unless no more tiles are available. The bag contains all remaining tiles to ensure the player draws his/her tile randomly. A player's hand is exposed so other competitors know which tiles are held. Therefore through a process of elimination the players can gain knowledge about the tiles that are left in the bag. More importantly, because of forced plays (see below), a player may take advantage of the tiles in another player's hand.

The tiles played must always form a simple connected region, and the colours of each tile played must match about their edges. This is the "golden rule" that holds for the duration of the game.

Forced Plays

An important concept in the game of Tantrix is forced plays. A forced play is available, whenever there is an empty space surrounded by three edges. At the beginning and end of each turn players must check all forced spaces to see if they can be filled. Once a player has made a forced play he/she must then pick up another tile (if available) and again check for possible forced plays. Therefore in Tantrix a player's turn can consist of multiple moves.

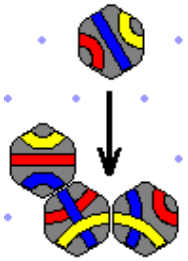

Figure 2: An example of a Forced Play (from Tantrix.com [9])

There are three other restrictions when playing a tile. These are outlined below.

1. You may not surround a forced space with another tile. A tile can only be played if it doesn't lead to a situation where it creates a space of four or more sides to be filled.

Figure 3: An invalid move due to restriction 1 (from Tantrix.com [9])

2. You cannot place a tile along a "controlled side" as it indirectly leads to the situation considered in the first restriction. When a forced space is created, it must be filled before playing more tiles along that row. For example in figure 2, position "A" is a forced space. Placing a tile in position "B" would create a string of forced moves indirectly leading to the situation presented in restriction 1. A preventative measure for figure 4 is to place the tiles across the controlled side in order from left to right, thereby ensuring the situation encountered in figure 3 won't occur at a later date.

Figure 4: A controlled side (from Tantrix.com [9])

3. You may not create a forced play where all three links are the same colour, as there is no tile that can fill this. An example is shown in Figure 5.

Figure 5: An invalid move due to restriction 3 (from Tantrix.com [9])

The Endgame

Once there are no tiles left in the bag all three of the restriction rules are lifted, leaving only the forced plays and the golden rule. This allows players to deliberately block the opponent's lines by creating spaces that are impossible to full.

Chapter 2: Classical AI Approaches

Game-Trees

A game tree is used to represent the different outcomes of choosing a move in a game. The tree is made up of nodes, where each node represents a possible game state or board layout. These nodes are connected together by edges to form the tree structure. Each edge represents the move required to achieve a resulting board layout. Game trees have been proven successful in managing the game information for smaller games; therefore they are commonly used in traditional AI algorithms. The game trees are generally large, thus storing and searching them effectively becomes an issue.

Static Evaluation Functions

Classical AI approaches generally rely on a static evaluation function to estimate the value of a leaf nodes game state in the tree. These values are then propagated up the tree to provide information on which game state is most likely to lead to a favourable result. The move that leads to the highest scoring game state can then be selected as the most promising.

Evaluation functions are considered static as they do not consider other nodes when calculating their value. Because of this, comparisons between nodes and the game states they represent can only be as good as the quality of their individually generated values.

Minimax

Minimax is an algorithm that is used in "full information" games; games in which the opponent's possible moves are known. It is used to allow comparisons of future game states beyond the next possible move. This is done by assuming the opponent's most favourable move is the least favourable for us. Thus, when estimating which move the opponent will make we can choose the one with the lowest score. The backpropagation strategy describes how this score is propagated up from the leaf. We can use a pessimistic approach and assume our opponent will always choose the move that is worst for us. This means that even if the opponent chooses a different move, our evaluation function would consider us to be in a better state than initially expected. Minimax can also be adapted to games with an element of chance (such as Tantrix) by using an averaging procedure at the chance nodes.

Alpha-Beta

Alpha-Beta is an extension to the search algorithm Minimax. It is used to search game trees and provide the same evaluation as Minimax but in a more efficient way. Minimax picks the best possible move by using information from an evaluation function to allow comparison between game states. In Alpha-Beta, pruning techniques are used to skip branches of the tree which are known to offer the opponent stronger moves than would otherwise be available.

Limitations

The large game tree in Tantrix limits the depth in which Minimax can search. Although Alpha-Beta does offer significant improvements over Minimax, it is still limited by the complexity of the game tree. Therefore, the evaluation function is required to accurately calculate scores for the leaf nodes. Finding an evaluation function in Tantrix that can perform these accurate calculations is difficult and, in this case, it is a major limiting factor for alpha-beta search programs. By instead using MCTS, we don't escape dealing with the complex game tree to some extent but instead choose which parts of the tree we explore or exploit.

Previous Work

In January 2005 the Tantrix playing bot, GoodBot, was first released. GoodBot is closed source and only a brief description of how it works has been released. This bot uses an alpha-beta search combined with an evaluation function that tries to predict the final length of the lines at the end of the game. In May 2005, a new open source bot named Oliver emerged. Oliver was written by Pieter Bolle and was submitted for a degree of Master of Engineering in at Katholieke Universiteit Leuven, in Belgium. Oliver works similarly to GoodBot by using a range of heuristics to score each game state. This was combined with an alpha-beta search to dethrone GoodBot.

Oliver's evaluation function works by predicting how long the line will be at the end of the game. Lines that might be connected are considered as a single line, modified by the probability that the right tiles will be drawn to connect these lines.

On May 11th Oliver beat GoodBot with a winning percentage of 64.5% to 30.5% (5% were tied). Three months later a new version of GoodBot was released to re-challenge Oliver. Two hundred games were played where GoodBot beat Oliver with a winning percentage of 55% to 39% (6% were tied). GoodBot has retained its title to this day and is considered the strongest Tantrix-playing bot.

Chapter 3: Monte-Carlo Tree Search

Overview

A major advantage of MCTS over traditional approaches is that MCTS doesn't rely on static evaluations for comparison. It instead works by using the results from a large number of simulated games to determine probabilistically the best move available. The MCTS cycle can be split into the four main parts outlined in the diagram below. This cycle can then be repeated a large number of times to increase the size and information in the game tree and thus provide more accurate grounds to predict the best solution.

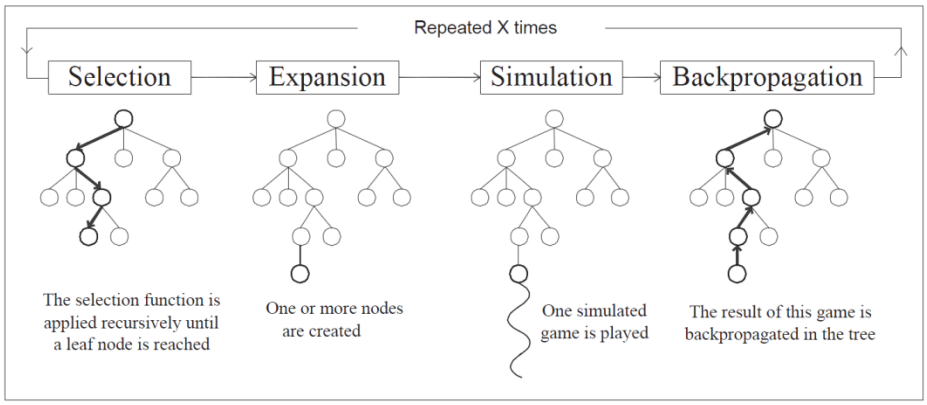

Figure 6: The MCTS cycle

In greater detail, the four parts of the MCTS cycle work as follows.

Selection

This is the process of traversing down known moves in the current game tree until a leaf node is reached. When traversing the game tree two strategies need to be taken into consideration; exploring a range of different moves to help decide which move is best (exploration), and exploiting the currently known best move to gain confidence that this move is actually favourable (exploitation). The UCT algorithm (discussed later) is used at this stage to find a balance between these two different approaches. Like the Minimax strategy each player's perspective is taken into account when choosing the move with the best UCT score. Once a leaf is reached, this stage ends and the expansion stage begins.

Expansion

An expansion strategy is used to add one or more additional nodes to the game tree. The simplest approach is to simply produce one node per iteration. This prevents the tree from growing too large too quickly and helps reduce the amount of memory required. Once the unexplored node has been added to the tree the simulation stage begins.

Simulation

The simulation process replaces the need for a static evaluation function. It involves playing a complete simulated game from the current leaf. This is done through self-play by choosing random moves for each player. Keeping the simulation stage simple allows for more simulations to be performed within a set timeframe. As the results are not intentionally biased towards any of the possible game situations, they would be expected to converge to a more even distribution as the number of simulations increases.

Backpropagation

A backpropagation strategy is used to propagate the result of the simulation back up the game tree. It does this by traversing back up the current path in the game tree, updating each node encountered on the way. The backpropagation strategy used informs each encountered node of the result from previous simulation (win, loss, or draw). This information is then used by the node to update its win rate. Each sibling of these nodes also gets updated to ensure its UCT score is kept current (see UCT below).

In practice the choice of the backpropagation strategy used makes little difference to the performance of the bot, especially when compared to the influences created by the other stages in the MCTS cycle. Therefore the backpropagation strategy seems more a matter of art and personal preference than an exact science.

Multi-Armed Bandit Problem

The key difficulty in MCTS is the implementation of the selection phase. To understand the issues involved, we can consider the multi-armed bandit problem. This problem draws its name from a traditional slot machine known as a one-armed bandit. When multiple levers are available, each can be associated with a win rate to model the chance of a win (pay-out) for that machine. The multi-armed bandit problem aims to maximise the return from these machines whilst trying to minimise potential losses.

Testing arms which don't pay off will result in a poor return, however, if not enough arms are explored there won't be enough confidence that better moves don't exist. This problem highlights the importance of balancing the exploration and exploitation of machines. Greedy algorithms try to exploit the best arms whilst often failing to explore other (possibly better) solutions.

We can use each arm from the multi-armed problem to represent a different game move that can be chosen. This allows us to find out which move would give us the best chance of winning by using each slot machine arm to represent a possible game move. Each time an arm is pulled some kind of reward is received, and in the case of the Tantrix game tree, this reward is either a win or a loss. Picking unexplored arms (exploration) improves the chance of finding a better move, but if this is repeated excessively, the confidence in each moves win rate will be low. However, if the same arm is frequently selected (exploitation), there won't be enough confidence that other less frequently explored moves won't transpire to be better.

Upper Confidence Bound for Trees (UCT)

Upper Confidence Bound for Trees (UCT) is an extension of MCTS that is used to manage the multi-armed bandit problem. UCT is implemented in the selection stage to maintain the balance between exploration and exploitation. This is done by creating a bias that slowly grows to favour the exploration of different paths in the game tree. The UCT score for each node is a combination of this exploration bias and the win rate of the node. The node with the largest UCT value is then chosen as the most suitable candidate for traversal.

The UCT value is comprised of an exploration term and an exploitation term. These terms are added together to make up the UCT value. The exploitation term is simply the win rate of the current node. This is calculated for each node by the ratio of games it won from the number of simulated games the node was involved in. Initially the UCT value was calculated from equation 7 until it was changed to a more customisable formula (equation 8) which includes a variable to customise the balance between exploitation and exploration.

In equations 7 and 8, the win rate is the ratio between number of wins the current node was involved and the number of times that node was visited (node.visits). The α value in equation 8 tunes the balance between exploration and exploitation. Through testing it was discovered that an α of 0.35 was the most successful for this program (see results section).

$$\text{UCT Value} = \text{winrate} + \sqrt{\frac{\ln(\text{parent.visits})}{(\text{node.visits})}} \quad [7]$$

The fundamental part of the exploration term is that it contains a ratio between the number of visits for the node and the number of visits for its parent. The exploration term for a node will get larger if the node's parent is visited but the node itself is not. This also decreases the exploration term when a node gets visited.

$$\text{UCT Value} = \text{winrate} + \alpha\sqrt{\frac{\ln(\text{parent.visits})}{(\text{node.visits})}} \quad [8]$$

Chapter 4: Design

System Design

When starting out this project I choose to start from an existing open source bot, named Oliver. Since I began from an existing system the Tantrix implementation could be ignored and the focus could remain on choosing the most promising move. The features inherited from Oliver include integration with the server Tantrix.com, graphical display of the game, and the representation details of the game and tiles.

Tiles

Each tile can be represented by a string of six letters, where each letter represents the colour at a particular side of the tile. The colours are listed in a clockwise direction, starting from the north-eastern side. Each tile also requires an orientation, a single number from zero to five, which shows how the tile has been rotated. The tile string represents each tile for the orientation 0 and the orientation increases by one for each single clockwise rotation of the tile. Additionally each tile is assigned a unique number (its position in a static array of tiles) which can also be used to represent it. For example, the tile in figure 9 is given the tile number 56 and the tile string *BGRGRB.*

Figure 9: The tile associated with number 56, the tile string BGRGRB, and the orientation 0 (from Tantrix [10])

Figure 10: The same tile with the orientation of 1 and an orientation of 3 (from Tantrix [10])

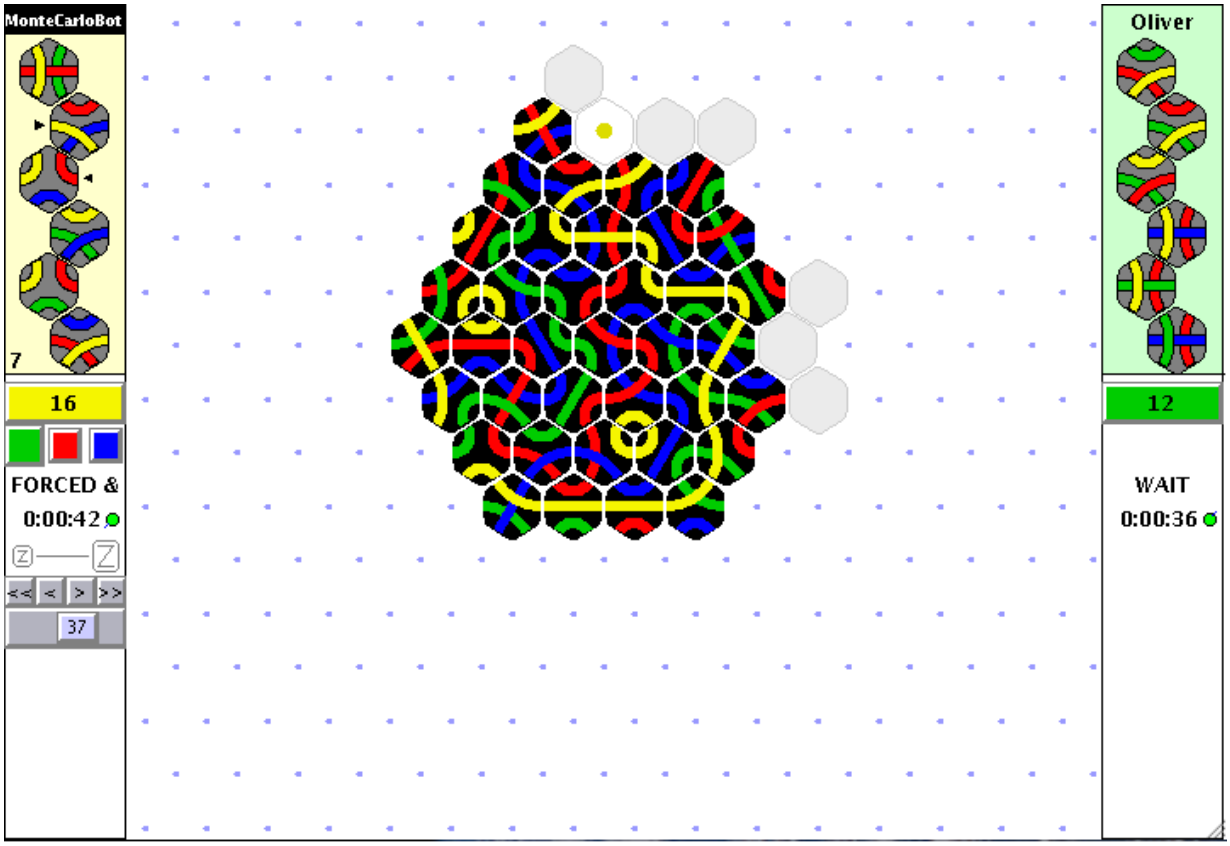

Figure 11: A screenshot of a game between the MCTS bot and an existing Tantrix bot, named Oliver.

Figure 11 shows a game between the MCTS bot, MonteCarloBot (MCB), and an existing Tantrix bot, named Oliver. The MCB is playing as the yellow player and Oliver is playing as the green player. Currently MCB is winning with a score of 16, this score comes from the length of the longest yellow line, which can be seen starting from the bottom left and ending at the top. The longest green line starts at the same tile (bottom left) and also ends near the top, but with a length of only 12. The time under each player shows the total number of seconds each player has spent thinking. As the number of tiles played by each player may not be equal, unless averaged over a large number of games, the displayed time is not a good indication of each bot's speed. The yellow dot at the top of the board highlights the position of a forced move and the colour of the player that can fill it. As it is currently MCB's turn it must fill this space.

Coordinates

Due to the hexagonal pieces in Tantrix, finding a simple coordinate system was a problem. The current system design uses a hexagonal grid with x and y coordinates, but in order to do this only half of the positions can be valid. Therefore x and y need to have the same parity. The neighbours of the position (x, y) in the grid can be reached by adjusting the x and y figures by the amounts shown in figure 12.

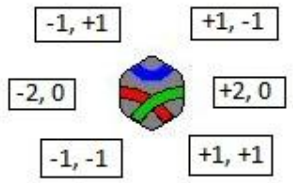

Figure 12: The adjustments on x and y respectively to move to a different coordinate in the coordination system. The tile is considered to be at coordinate (x, y).

Moves

By using these representations a move can simply consist of a tile number, an orientation, and a coordinate. Once a move has been made, if the tile bag is not empty, a new tile is drawn and added to the player's stack to replace the previously played tile.

Monte Carlo Bot Design

Tantrix Game Tree

At the start of each turn the game tree is reset and a new one is created. Ideally the root would be adjusted so it always represents the current board situation. This would save the relevant results from previous simulations. However, in Tantrix this makes little difference, as a turn can consist of multiple moves. To find the current game state, the game tree may have to be searched deeply after an opponent's turn. Even if the game state is found, the simulations can be quickly repeated so changing the root becomes redundant. Therefore the tree is simply wiped and the MCTS process is started fresh.

Tantrix is different to most other games in that a turn can consist of multiple moves. There are three main parts to each turn; playing all available forced moves, selecting a free move, and then playing any remaining forced moves. There is an element of chance in Tantrix due to drawing a random tile after a tile is played. As a result, the same path in the tree cannot be expected to remain valid between simulations. A new type of node called a chance node was introduced to deal with this issue.

Chance Nodes

The representation of chance nodes in a game tree has proven to be a successful addition for traditional AI game playing models involving games with a chance factor, such as Backgammon[13].

The chance nodes used in Tantrix always follow a game node (traditional node) and only represent the last tile drawn from the bag. In the game tree, traveling to game nodes can be thought of as playing a move. The following chance node would represent the tile that was drawn as a result. If the tile drawn is contained in one of the chance nodes then that path through the tree is taken. Otherwise, a new chance node containing this move is added to the tree and the expansion stage begins.

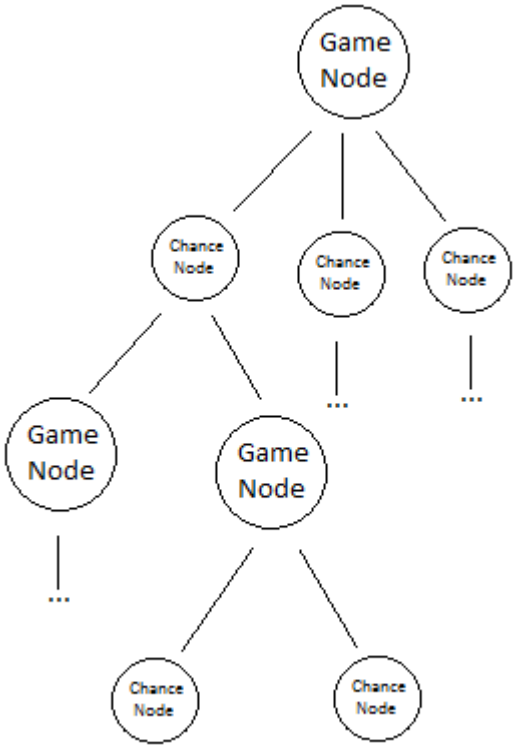

Figure 13: A game tree with the inclusion of chance nodes. At each game node all possible forced moves must be made before and after each free move is made. Therefore multiple game nodes can be visited before changing player.

Best Move Chooser

After the simulations are completed all possible moves are compared and the move with the highest visit count is chosen (as opposed to the move with the best win rate). This is because choosing the move with the best win rate doesn't consider the number of simulations that move was involved in. Therefore the confidence in that win rate could be low. Since UCT often exploits the most promising moves, they become the most visited nodes in the tree. This allows a nodes visit count to indicate the lower bound of its win rate. In doing so, our confidence for each move is considered, allowing moves to be chosen that do not necessarily have the highest win rate, but the win rate with the highest lower bound.

In practice it is reasonably uncommon to have a situation in Tantrix where the move with the highest win rate is not the most visited move. Therefore the performance gained/lost from choosing these different approaches is small when in comparison to the influences from factors such as alpha.

Monte Carlo in Tantrix

Like most games, the number of moves required to finish a simulation in Tantrix becomes smaller throughout the game. Since the MCTS bot is run for a set amount of time per move, fewer iterations of the MCTS cycle are performed at the start of the game. This causes the number of cycles to increase towards the end of the game, often by over 2000%. Throughout the game the number of possible tiles to be drawn decreases. This makes it easier to predict the end game result, so the MCTS cycles towards the endgame contain the most valuable information. Increasing the number of these cycles towards the end of the game radically increases the information known about the game's end state. This allows the bot to choose stronger moves. As there are fewer moves left in the game the opponent is left with less time to recover.

This approach also suggests weaker moves may be chosen at the start of the game. In Chapter 6, a method has been suggested to test whether this has much of an influence on the performance of the bot.

Selection

The game tree is traversed until a leaf node is encountered. The decision in the selection stage acts like a Minimax algorithm with a depth of one. It simply searches from the root by iteratively selecting the child with the highest UCT score. As the win rate for the opponent's move was reversed in the backpropagation stage, when choosing the child to explore, the one with the highest UCT value can always be chosen. This automatically considers the relative player's perspective while choosing their best move.

Expansion

In the expansion stage all possible moves are added as new children, as each node contains no information on the possible moves available. These nodes are initialised with their parent's win rate as this is currently the best estimate we have of their true value. Once the node has been visited the stored win rate for the node becomes its own.

Simulation

During the simulation stage a complete simulated game is played from the current leaf. Random moves are chosen for each player to ensure the play out of moves is fast and unbiased. Each simulated game returns a value {1, 0, 0.5} to represent whether the game (from the MCTS bot's perspective) was won, or lost, or drawn.

Support for Draws

During the simulation stage any draw is considered as half a win, this allows them to be distinguished from wins and losses and still contribute information to the win rate.

Tantrix sometimes ends up in a stuck game state, where neither player has an available move. When this situation arises the game is considered a draw. This scenario can be exploited using the MCTS approach as each simulation that reaches the stuck state has a win rate of 0.5. If the nodes' siblings are considered to have a lesser win rate then a draw becomes favourable.

Backpropagation

Each move is undone and the simulated game result is passed on to the nodes involved. The UCT scores for the move and all its siblings are also calculated. All nodes involved are updated with the resulting score, as calculated from the method shown in Figure 14.

```
public void upDateUct() {
    double explorationTerm = 0;
    int lastGameNodeVisits = getParent().getParent().getVisits();
    explorationTerm =  alpha*Math.sqrt(Math.log(lastGameNodeVisits)/(visits+1));
    uct = getWinrate() + explorationTerm;
}
```

Figure 14: The method that is called to update the UCT values for sibling nodes.

The win rate for a move is always considered from the point of view of the player that made it. Every simulation we win is considered a loss from the opponent's perspective. This simplifies the selection process as the maximum win rate can be used to represent the best move for both players.

Each tile has the same chance of being drawn from the bag. Therefore, chance nodes contain no interesting information about the odds of drawing tiles compared to others. The chance of drawing required tiles will influence the win rate as the tiles are drawn randomly. The backpropagation process was simplified to exclude chance nodes completely by passing information directly between game nodes.

There are other possible ways to involve chance nodes in the backpropagation stage but as they appeared to offer no major benefits I chose to inform each game node of its result whilst traversing back up the tree. An alternative approach is setting chance nodes to contain the average win rate of their children. Each remaining tile has an equal chance of being drawn. Due to this, the average win rate of the children would represent the average score expected when traveling through this node. As the win rate is already influenced by the chance to draw tiles, this process is not necessary.

Chapter 5: Results

The MCTS bot was tested against GoodBot, the currently best performing bot on Tantrix.com. Games were played against different versions of the MCTS bot. The versions in Figure 15 were used to test the effect of changing the alpha value. For this test each MCTS bot was allowed a fixed amount of time per move (four seconds). Figure 16 shows the results of keeping alpha fixed (at 0.5) and allowing each MCTS bot a different amount of time per move. The results shown in both Figure 15 and Figure 16 are against GoodBot and include error bars for a 95% confidence interval. The number of trials for each bot can be found in Appendix A.

Figure 15 shows a win rate of over 50% against the currently best performing bot, GoodBot, for every alpha value tested within the range of 0.35 and 1. This was done using a conservative amount of time per move (four seconds). Even in the worst case this falls well within the maximum time allowed per game (15 minutes). Increasing the time the bot is allowed for each move further increases the win rate (as shown in Figure 16).

Three hundred and forty games were played out against both GoodBot and Oliver. In these games an alpha of 0.5 was used with a maximum of 6 seconds allowed per move. The win rate against Oliver was 71.8% and the win rate against GoodBot was 68.2%. This shows that the MCTS bot comfortably outperforms what were considered to be the two best Tantrix playing bots.

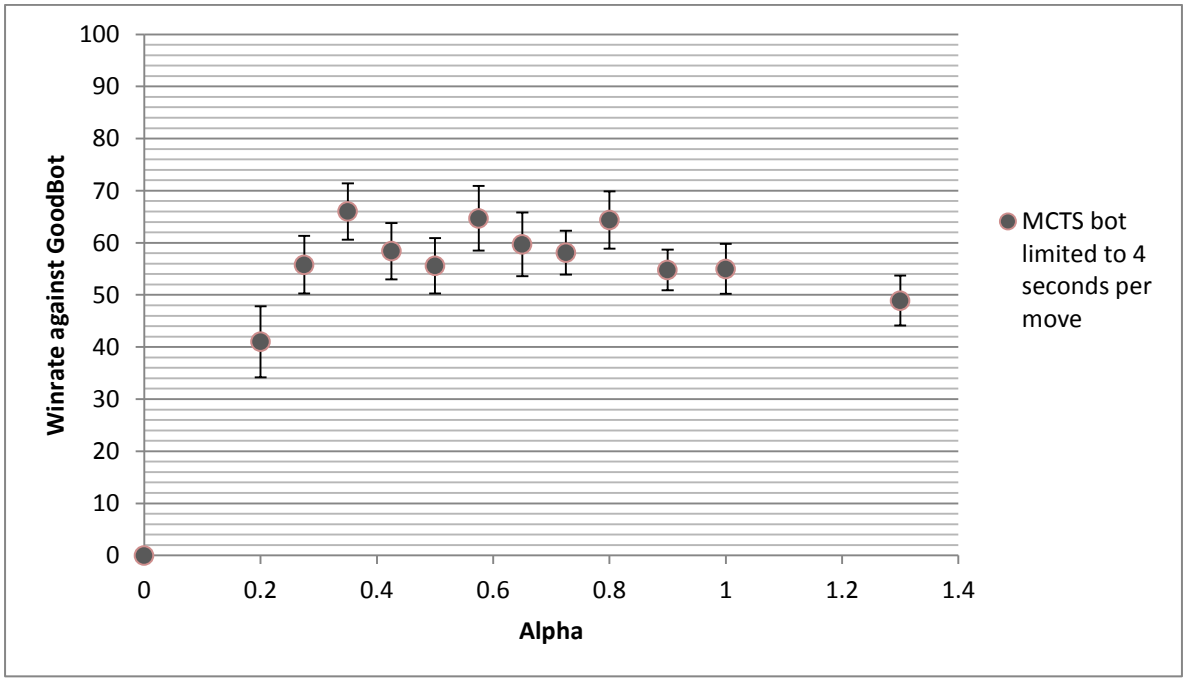

Figure 15: Showing the effect of changing alpha on the win rate against GoodBot using a 95% confidence interval. Note: as alpha 0 did not manage to win a game calculating a confidence interval for this value was not possible

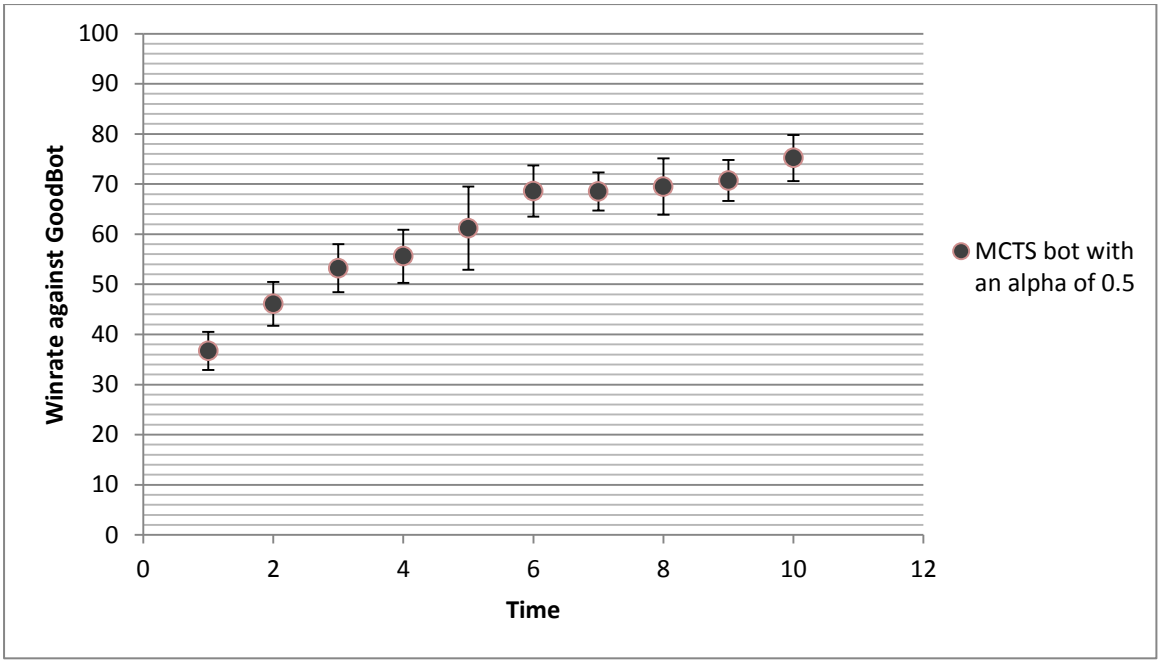

Figure 16: Showing the effect changing the time allowed per move has on the win rate when against GoodBot with a 95% confidence interval

Discussion

Alpha

The alpha value plays an important part in the UCT formula as it can be used to directly influence the balance between exploitation and exploration. An alpha value of 0 would be a purely greedy approach and only explore the move with the highest current win rate. In particular the first simulated move leading to a win will always be chosen, although the optimal value for alpha is likely to be unique for the program. Through testing it was found that values below 1 provided the best results. More precisely it was found that an alpha value of 0.35 created the best performing bot. A low alpha causes the UCT equation to favour exploitation rather than exploration. As the game tree for Tantrix is complex the game nodes generally have a large number of siblings. Increasing the number of simulations involving the favourable moves becomes important as it increases the confidence in the moves that are most likely to be played.

Time allowed each move

In Tantrix the total decision time allowed for each player is 15 minutes, any longer and the game is considered a loss. As there is a maximum of 50 possible moves for a player (one player draws all of the tiles), up to 18 seconds can be used to play each move.

As the time allowed for each move increases, the bot can perform more simulations. This provides more accurate win rate estimates, while also allowing better traversal in the selection stage.

Environment

All results were calculated using the following installed software and hardware.

Software

Operating System:	Mac OS X 10.6.4
Java Version:	1.6.0_17

Hardware

Processor:	2.66GHz Intel Core 2 Duo
Memory:	4GB 1067 MHz DDR3

Storing & Retrieving Game Results

As each game ends, the game information is saved into a text file. This information includes the players, the scores, the winner, and the date of the game. These results were processed by another program whose primary job was to display the bots estimated win rate along with the corresponding accuracy and confidence margins (as explained in the section below).

Confidence

The observed win rate when the Monte Carlo bot plays against GoodBot is our best estimate of the bots actual advantage or disadvantage over GoodBot. However, because neither program gains any knowledge from previous games, each game can be considered as a single random trial between the two programs. Therefore we can produce confidence intervals for the true win rate using the observed data. The standard formula for a 95% confidence interval (using a normal approximation to the binomial distribution) after n trials is:

$$\bar{p} \pm 1.96\,\bar{\sigma} \qquad [17]$$

Where,

$$\bar{p} = \frac{wins}{n} \qquad [18]$$

And,

$$\bar{\sigma} = \sqrt{\frac{\bar{p}(1-\bar{p})}{n}} \qquad [19]$$

The significance of 95% confidence in an interval is that if the true win rate were to lie outside of this interval, then results as extreme as those we observed would have occurred less than 1 time in 20.

Chapter 6: Future Work

The MCTS bot plays weaker moves at the start of the game because fewer simulations can be performed. To test the impact this has on the final results, a new version of the MCTS bot could be created that simply plays randomly for the first half of the game. If this new bot performs similarly to the original MCTS bot (against a common opponent), it can be concluded that the early stages of the game provides little influence towards the end results.

A fast approach to simulate games is to simply choose moves randomly in the selection stage. When the number of simulations is small the win rates often become a misleading representation. By adding expert knowledge to the game tree simulated games can better represent played moves. This leads to more relevant game outcomes which can improve the decisions made if few simulations are performed. Although adding expert knowledge has been found to be a successful optimisation for MCTS[14], it has many disadvantages that make it impracticable for Tantrix.

The three main disadvantages for using expert knowledge in Tantrix are:

- It relies on a static evaluation function;
  - Reliable evaluation functions for Tantrix are hard to find.
- It greatly reduces the number of simulations that can be performed in the same timeframe;
  - This means there is less information to calculate win rates from.
- It can make the bot exploitable;
  - Players can recognise patterns on what the expert knowledge thinks good moves are and can use these decisions to their advantage.

The focus of this report was on the more strategic two player version of Tantrix. This leaves an opportunity for future projects to investigate the use of MCTS in three and four player games and extend this project to cover these versions.

If multiple nodes each have a win rate of 100% they are all explored equally. The move chosen is generally the first of these encountered. A future improvement could to use a simple evaluation function at this point to pick the move that maximises the score difference. This will cause the Monte Carlo bot to try and win by the largest margin possible in the final moves of the game.

Although the time limit per game is fifteen minutes, the bot's decision time was greatly restricted to speed up gameplay and make it a more enjoyable opponent. It was shown in the results section that the bots performance can be improved by increasing the time it has for each move. The same performance increase could be expected from making the bot faster, as more simulations could be performed per turn. This improves the accuracy of the estimated win rate for each node and provides the selection stage more choice on which parts of the tree to explore.

Parallel support

Some ways to provide parallel support include:

- Running multiple simulations in parallel
  - The results from multiple simulations can then be backpropagated back up the tree
- Running the whole tree-traversal and simulation process in parallel on a shared tree
  - Allows more nodes to be visited than simply running the simulations in parallel

    - Each traversal relies on results from previous traversals so it is different to the sequential version.
- Running the whole tree-traversal and simulation process in parallel on individual trees
    - Results can be combined when it is time to make a decision
    - Performs the same as sequential MCTS.
- The paper by Rimmel[7] has already looked into these different approaches and concluded that the overhead from sharing tree data is too great to offer any substantial improvements to the results without altering the algorithm itself.

Unsolved Problems

The focus of this project was to test to see if MCTS could be a viable alternative to the current Tantrix bot designs. Customising the Tantrix system itself and how it stores and finds information was left for future work; because of this some problems were inherited from the existing implementation. Since the Tantrix system does not have a continuous playing surface, a long string of tiles played in the same direction can occasionally play off the end of the table. This problem occurs rarely but once encountered it causes an error to be thrown during the calculation each player's score. At this point the automatic playing of games stops to allow the scores to be verified by a human. To avoid the possibility of ever meeting this exception, future projects need to create their own implementation of the game.

After running the MCTS bot through a profiler it can be seen that a large amount of time is spent searching for free and forced moves. Currently the Tantrix system does this by brute force. The outline of the board is examined and every available tile in every orientation is tested to see whether it could fit. This needs to be done for every simulated move in every simulation and becomes the major bottleneck in the MCTS bot. An alternative approach would be to update and reuse the previous list of available moves. This reduces the number of moves that need to be recalculated to moves residing in a local area of the board, thus speeding up the process considerably, and allowing the MCTS bot to play out more simulations per move.

Chapter 7: Conclusion

Following the success of MCTS in the game of Go, this report set out to see if MCTS could be a viable algorithm for the game of Tantrix. Not only did MCTS prove to be a viable algorithm it out performed all existing designs. MCTS was implemented into a Tantrix playing bot to control how its moves are chosen. This bot was then played against the best performing bots on the Tantrix server (Oliver and GoodBot).

The MCTS bot played 340 games against both Oliver and GoodBot using an alpha of 0.5. Even though the MCTS bot was restricted to only 6 seconds per move it still out performed both of its opponents. Against Oliver the MCTS bot won 229 games (only losing 90) to give it a win rate of 71.8%. The same MCTS bot won 214 games against GoodBot while losing 100 games (a winning percentage of 68.2%).

These results show that by implementing MCTS into a Tantrix playing bot considerable performance improvements were noticed over the existing designs. It was found that MCTS can be used successfully as an alternative to static evaluation functions and with minimal changes this algorithm can be effectively adapted to suit complex games such as Tantrix.

This project further demonstrates the potential of MCTS, and helps show the flexibility of the algorithm by verifying another game that MCTS can be successfully applied to. I feel this project is a step in the right direction to creating a more universal MCTS algorithm that can play successfully in a range of different games without any changes to the MCTS algorithm itself. I hope that this work will provide a good base for the future development of Tantrix playing bots and can be used towards creating a more general game playing MCTS strategy.

Bibliography

[1] Bolle, P. (2004-2005). Design and realisation of a program for playing tantrix. Thesis for Master of Engineering at Katholieke Universiteit Leuven.

[2] Bolle, P. Program 'Oliver'. (15 April 2009). Retrieved July 03 2010,from Sourceforge: *http://tantrix-oliver.sourceforge.net/*

[3] Chaslot, G., Bakkes, S., Szita, I., & Spronck, P. (2008). Monte-Carlo Tree Search: A New Framework for Game AI. Maastricht University. *http://sander.landofsand.com/publications/Monte-Carlo_Tree_Search_-_A_New_Framework_for_Game_AI.pdf*

[4] Kocsis, L., & Szepesvari, C. (2006). Bandit based monte-carlo planning, from Machine Learning: ECML vol. 4212 (2006), pp. 282–293.

[5] Matsumoto, S., Hirosue, N., Itonaga, K., Yokoo, K., & Futahashi, H. (2010). Evaluation of Simulation Strategy on Simgle-Player Monte-Carlo Tree Search and its Discussion for a Practical Scheduling Problem. Proceedings of the International MultiConference of Engineers and Computer Scientists Vol III.

[6] Pinto, P. Minimax Explained. (2002, July 28). Retrieved July 03 2010, from AI-depot, *http://ai-depot.com/articles/minimax-explained/*

[7] Rimmel, A. (2009, September 15). Improvements and Evaluation of the Monte-Carlo Tree Search Algorithm. *http://tao.lri.fr/Papers/thesesTAO/RimmelPhD.pdf*

[8] Takeuchi, S., Kaneko, T., & Yamaguchi K. (2008). Evaluation of Monte Carlo Tree Search and the Application to Go. CIG08, pages 191—198.

[9] Tantrix. (2009, September 22). Retrieved July 1, 2010, from Tantrix*: http://www.tantrix.com.html*

[10] Tantrix bot starter kit documentation. Retrieved July 02 2010, on request from www.tantrix.com.

[11] UCT. (26 October 2009). Retrieved July 01 2010, from Sensei's Library: *http://senseis.xmp.net/?UCT*

[12] Van den Broeck, G., Driessens, K., & Ramon, J. (2009). Monte-Carlo Tree Search in Poker using Expected Reward Distributions. *http://www.springerlink.com/content/5118833l5764476m/fulltext.pdf*

[13] Van Lishout, F., Chaslot, G., Uiterwijk, J. Monte-Carlo Tree Search in Backgammon, from Computer Games Workshop, pp. 175–184 (2007) *http://www.montefiore.ulg.ac.be/~vanlishout/publications/vanlishout_backgammon.pdf*

Appendix A

Source Code for MCTS

Simulation Method

```
/*
 * Will simulate rest of the game and
 * returns a double indicating if the game was won by the current player
 * 1 - the game was won by us
 * 0 - the game was lost by us
 * 0.5 - the game was a draw
 * Currently a draw is considered as a loss
 * by the player specified in the arguments
 */
public double runSimulation() {
    //  System.out.println("starting simulation");
    double won=0.5;//0 for loss, 1 for win, 0.5 draw
    boolean gameStuck = false;
    boolean hasPlayedFreeMove = $game.hasPlayedFreeMove();
    int i=0;
    Move move = null;
    int playerNo =-1;
    Stack<Move> moves= new Stack<Move>();
    Stack<Integer> movesMadeBy= new Stack<Integer>();
    while (!$game.isFinished() && gameStuck==false){
        try{
            move = useRandomMove();
            if(move != null){
                moves.push(move);
                movesMadeBy.push($game.getPlayerNumber());
                $game.playMove(move.getTileNr(), move.getX(), move.getY(), move.getRotation());
                i=0;
            }else{
                i++;
                System.err.println("end turn for player\n");
                getGame().changePlayer(); // It's now the turn of the other player
                if(i>2){
                    gameStuck = true;
                    System.out.println("neither player can play current simulation - simulation considered a draw");
                }
            }
        }  catch (IllegalMoveException e) {
            System.err.println("ILLEGAL MOVE OCCURED!!");
            e.printStackTrace();
        } catch (Exception e){
            System.err.println("PROB TRIED TO PLAY A NULL MOVE");
            System.err.println("There Are "+$game.getTileBag().getNbRemainingTiles()+"tiles in the bag");
            System.err.print("MOVE PLAYED: ");
            System.err.println(move);
            e.printStackTrace();
            System.exit(0);
        }
    }
    if(gameStuck==false){
        $game.calculateScores();
        int score1 = $game.getRobotPlayer().getScore();
        int score2 = $game.getOtherPlayer().getScore();
        if (score1>score2){
            won = 1;
        }else if(score1<score2){
            won = 0;
        }else{
            won = 0.5;
        }
    }
    //undo the moves
    while(!moves.empty()){
        move = moves.pop();
        playerNo = movesMadeBy.pop();
        $game.changePlayerAfterUndo(playerNo);
        undoMove(move);
        move.won(won);
    }
    $game.setPlayedFreeMove(hasPlayedFreeMove);
    //System.out.println("finshed simulation, We Won: "+won);
    return won;
}
```

Selection, Expansion and Backpropagation Method

```
public void simUsingTree()throws SearchTerminatedException, IllegalMoveException{
    int originalTileNo = $game.getLastDrawnTileNo();
    Stack<Integer> movesMadeBy= new Stack<Integer>();
    boolean hasPlayedFreeMove = $game.hasPlayedFreeMove();
    //SELECTION
    GameNode n = $game.getTree().getRoot();

    n.visited();//root visits used for total simulations

    if(!n.childrenContain($game.getLastDrawnTileNo())){
        $game.getTree().insert(n, new GameNode($game.getLastDrawnTileNo()));
    }
    n = n.getChildThatContain($game.getLastDrawnTileNo());

    while(n.hasChildren()){
        n = $game.getTree().getUctChild(n);
        n.visited();
        movesMadeBy.push($game.getPlayerNumber());
        $game.playMove(n);

        if(!n.childrenContain($game.getLastDrawnTileNo())){
            $game.getTree().insert(n, new GameNode($game.getLastDrawnTileNo()));
        }
        n = n.getChildThatContain($game.getLastDrawnTileNo());
    }
    if(n.hasChildren()){
        System.err.println("Error - SHOULDNT BE CHILDREN HERE");
    }

    /* EXPANSION (adds all possable moves to tree) */
    if(!$game.isFinished()){
        Move[] moves = getMoveGenerator().getForcedMoves(getGame());
        if(moves.length==0){
            moves = getMoveGenerator().getFreeMoves(getGame());
        }
        if(moves.length!=0){
            for(int i=0;i<moves.length;i++){
                $game.getTree().insert(n, new GameNode(moves[i]));
            }
        n = $game.getTree().getUctChild(n);
        n.visited();
        movesMadeBy.push($game.getPlayerNumber());
        $game.playMove(n);
        if(!n.childrenContain($game.getLastDrawnTileNo())){
            $game.getTree().insert(n, new GameNode($game.getLastDrawnTileNo()));
        }
        n = n.getChildThatContain($game.getLastDrawnTileNo());
        }
    }
    double won = runSimulation();//simulation
    $game.setLastDrawnTileNo(originalTileNo);

    /* BACKPROP */
    do{
            n = n.getParent();
        int player = movesMadeBy.pop();
        $game.changePlayerAfterUndo(player);
        undoMove(n.getMove());
        if(player !=$game.getRobotPlayerNo() && won !=0.5){//if player is not us reverse win/loss for their perspective
            if(won == 1){
                n.update(0);
            }else if(won==0){
                n.update(1);
            }
        }else{
            n.update(won);
        }
        n = n.getParent();;
    }while(movesMadeBy.size()!=0);
    $game.setPlayedFreeMove(hasPlayedFreeMove);
    $game.calculateScores();//reset scores after simulations
}
```

Appendix B - Detailed Results

alpha=0.5 Time=10s VERSE GoodBot
wins = 234
lost = 77
played = 333
Win rate = 75.2 +/- 4.6
---
alpha=0.5 Time=1s VERSE GoodBot
wins = 208
lost = 358
played = 611
Win rate = 36.7 +/- 3.8
---
alpha=0.5 Time=2s VERSE GoodBot
wins = 214
lost = 250
played = 503
Win rate = 46.1 +/- 4.4
---
alpha=0.5 Time=3s VERSE GoodBot
wins = 200
lost = 176
played = 417
Win rate = 53.2 +/- 4.8
---
alpha=0.5 Time=5s VERSE GoodBot
wins = 74
lost = 47
played = 134
Win rate = 61.2 +/- 8.3
---
alpha=0.5 Time=6s VERSE GoodBot
wins = 197
lost = 90
played = 312
Win rate = 68.6 +/- 5.1
---
alpha=0.5 Time=7s VERSE GoodBot
wins = 354
lost = 163
played = 560
Win rate = 68.5 +/- 3.8
---
alpha=0.5 Time=8s VERSE GoodBot
wins = 162
lost = 71
played = 257
Win rate = 69.5 +/- 5.6
---
alpha=0.5 Time=9s VERSE GoodBot
wins = 311
lost = 129
played = 478
Win rate = 70.7 +/- 4.1
---
alpha=0 Time=4s VERSE GoodBot
wins = 0
lost = 53
played = 53
Win rate = 0.0
---
alpha=0.2 Time=4s VERSE GoodBot
wins = 71
lost = 102
played = 204
Win rate = 41.0 +/- 6.8
---
alpha=0275 Time=4s VERSE GoodBot
wins = 158
lost = 125
played = 309
Win rate = 55.8 +/- 5.5
---
alpha=0.35 Time=4s VERSE GoodBot
wins = 175
lost = 90
played = 301
Win rate = 66.0 +/- 5.4
---
alpha=0.425 Time=4s VERSE GoodBot
wins = 173
lost = 123
played = 321
Win rate = 58.4 +/- 5.4
---
alpha=0.5 Time=4s VERSE GoodBot
wins = 174
lost = 139
played = 341
Win rate = 55.6 +/- 5.3
---
alpha=0.575 Time=6 VERSE GoodBot
wins = 132
lost = 72
played = 226
Win rate = 64.7 +/- 6.2
---
alpha=0.65 Time=4s VERSE GoodBot
wins = 138
lost = 93
played = 251
Win rate = 59.7 +/- 6.1
---
alpha=0.725 Time=4s VERSE GoodBot
wins = 272
lost = 196
played = 518
Win rate = 58.1 +/- 4.2
---
alpha=0.8 Time=4s VERSE GoodBot
wins = 177
lost = 98
played = 287
Win rate = 64.4 +/- 5.5
---
alpha=0.9 Time=4s VERSE GoodBot
wins = 322
lost = 266
played = 638
Win rate = 54.8 +/- 3.9
---
alpha=1.0 Time=4s VERSE GoodBot
wins = 193
lost = 158
played = 379
Win rate = 55.0 +/- 5.0
---
alpha=1.3 Time=4s VERSE GoodBot
wins = 194
lost = 203
played = 424
Win rate = 48.9 +/- 4.8
---
alpha=0.325 Time=6 VERSE GoodBot
wins = 112
lost = 57
played = 184
Win rate = 66.3 +/- 6.8
---
alpha=0.325 Time=6 VERSE Oliver
wins = 170
lost = 58
played = 245
Win rate = 74.6 +/- 5.5
---

alpha=0.5 Time=6 VERSE
GoodBot
wins = 214
lost = 100
played = 340
Win rate = 68.2 +/- 5.0
---
alpha=0.5 Time=6 VERSE
Oliver
wins = 229
lost = 90
played = 340
Win rate = 71.8 +/- 4.8
