A New Method of Averaging in Monte Carlo Game Tree Search

Matthew McKay

Poolesville High School

Professor Clyde Kruskal

University of Maryland, College Park

Abstract

In recent years, the technique of Monte Carlo Game Tree Search (MCGTS) with the upper confidence bounds applied to trees algorithm has been developed for game playing computer programs, such as Go. A component of the technique involves taking a weighted average of a node's children in the tree. This paper shows how the average can be improved by using a generalized average in the place of the weighted average. Testing of the generalized average compared to the normal average on a Bridge program using MCGTS showed that it is better at predicting the number of tricks taken, however it is not better at selecting the right card to play. Using the MCGTS could lead to better game programs for Bridge and other games that use similar algorithms for their artificial intelligence.

Introduction

In the last few years a new technique for game playing computer programs has been developed. It uses the Monte Carlo Game Tree Search (MCGTS) with the Upper Confidence bounds applied to Trees (UCT) algorithm, which is how the best Go programs work. It is suggested that this technique may be improved by using generalized averaging in a component of the method instead of the usual weighted average. The goal of this project is to study the averaging method in UCT and improve upon it by using this new method of averaging.

The game of Bridge is used as a vehicle to test the averaging method. The main reason for this is that, unlike Go, the actual values attributed to certain moves can be calculated using a Double Dummy Solver (DDS). For Go, there is no efficient method to determine an optimal move. Not only that, but a specific play of a hand in Bridge has a resulting value based on how many tricks a pair takes, while Go is merely a win/lose situation. This allows the error in estimation from the actual values to be calculated, and the resulting error represents something meaningful. Also, the play of a Bridge hand is relatively simple to program. Since no one has applied UCT to Bridge before, the project is also observing the advantages to using UCT to estimate the values for a Bridge hand instead of calculating them using a DDS. The importance of the project is twofold: First, the algorithm seems to improve upon the standard MCGTS with UCT, and second, if the program can be made significantly faster than current DDS's, it could possibly improve the current Bridge programs.

After testing a set of 100 hands, the result was not exactly what was expected. While the new averaging method with higher powers and constants does have a lower error, as was predicted, it is not significantly better than the normal average with the same constant, in terms of selecting the correct card. The higher constants may only be beneficial in predicting how many tricks will be taken. Also, the extra cost of computing the generalized average may make the technique impractical.

Go

Go, an ancient strategy game for two players, is unlike most other board games in that computers are not very good at playing it, at least not nearly as good as humans. With games such as Go or chess, the simplest way a computer could make a decision about where to move would be to map out every possible board position in a tree. The root of the tree branches off into multiple children each of which branch off to more children, eventually reaching a node with no children, known as a leaf node. In the game tree for Go, the root represents the initial board position, every branch off of that is a new board position resulting from one of the many possible moves, every branch off of those branches is another board position involving another possible move from the previous position, and so on down the tree. The leaves represent a completed game, with the branch leading to them holding all of the moves made to get there. The computer could, in principle, select the optimal move by mini-maxing the leaf values up to the root. Since Go is a perfect-information game, where every move is known, this would be a simple task to perform. However, the problem with Go is that the game tree is much too large. This is due to the fact that this tree grows exponentially with each additional level down in the tree. There are many moves to make each turn (19^2 to begin with), and the tree maps out those moves all the way through the game, which would result in an enormous tree. To avoid this, the best Artificial Intelligence (AI) programs for Go make a decision a different way, using Monte Carlo methods, more specifically the UCT algorithm, which is based on the principles of Monte Carlo methods (Gelly). Programs that use this method, while much better than previous programs, are still unable to beat the best human players.

Monte Carlo Game Tree Search

The way Monte Carlo Game Tree Search (MCGTS) with Upper Confidence bounds applied to Trees (UCT) works is that it plays out may random games from the initial position. As it plays through the random games, a tree is created for every move it makes, mapping out the possible moves like in the previously mentioned full game tree. However, the MCGTS does not create the entire tree of possible moves, as that would defeat the purpose. It instead creates a tree only of the moves made in the randomly played games. This makes the method incredibly fast, and still allows for a reasonable approximation of the true value. By taking the values determined for each of the random games, which are the values stored in the leaf nodes, and recursively averaging them up the tree into the parent nodes, whose values are the average of their children's values, the resulting value is the final value of the root.

The UCT algorithm carefully selects which children to evaluate as it randomly plays a game. The paper written by Gelly and Wang explains this algorithm and its application to Go, as well as how it was used in one of the current best Go playing programs, MoGo (Gelly). While the UCT algorithm is a form of Monte Carlo methods, it is not uniformly random. As the algorithm runs through more games, it slowly creates a tree of moves for the games it plays, where each node in the tree stores the value for a certain move, as well as how many times that node has been visited so far. It then uses this tree to decide what moves to make, balancing between moves that have previously resulted in good values and moves that have not yet been visited much. The easiest way to impact this is to modify the constant in the function used. Higher constants cause the algorithm to visit infrequently visited nodes more often, while lower constants have the algorithm continue to visit the better moves more often. The compromise allows for obscure plays to be found yet still let the best moves be emphasized.

Every time that the method is run, it goes through one play of the game, using the tree to decide what moves to make. When a new path is taken in the tree, a child is made for every possible next move that could be made after the most recent move. This means that the tree starts out empty, creating children as it plays through the first time. Each of the nodes in the tree stores a value and a number of visits. In the beginning, to decide which path to take, the algorithm first selects a move at random from the possibilities, but it only selects from the moves that have not been played before (or whose number of visits is zero). However, if each of the next moves has been played once, then a function is used to determine which move to make next. The selected move is the one whose node maximizes (or if one of the opponents' turns, minimizes) the function:

$$V_i = X_i + \sqrt{\frac{c \ln n}{T_i}}$$

Where X_i represents the value of the i^{th} child, c is a constant (to be selected), n is the total number of times that the current node has been visited, and T_i is the number of times that the i^{th} child has been visited (see Rajkumar, e.g.). The X_i part of the function makes the algorithm generally select the best moves more often. The expression that is added to X_i is what allows the algorithm to explore other options besides the best, because as the number of times the parent node is visited increases, the value of the expression $\sqrt{\frac{c \ln n}{T_i}}$ grows at a logarithmic rate. Even moves that seem bad will have a greater function value with the addition of the expression $\sqrt{\frac{c \ln n}{T_i}}$ than the moves that seem better, causing the algorithm to try those to search for other, possibly better seeming moves. If it is not successful, then the value of the expression $\sqrt{\frac{c \ln n}{T_i}}$ decreases, causing the algorithm to go back to searching the better moves more. However, if it is successful, then those moves are searched more, leading to a more accurate value. After playing through a game, the algorithm goes back up the tree, calculating a new value for each parent node. The new values are just the weighted mean of the children's values, which are the values of the children weighted by the number of visits. By using a weighted average, the moves that are better and tend to be visited more have greater weight, and therefore the resulting value is closer to them, as in Figure 1. The advantage of this is that even though other paths are explored, the values resulting from them have little impact on the final value. The MCGTS can play out large numbers of games, going through the tree once each time, to get a reasonably accurate value for the game, which is the value stored in the root after running through all of the games.

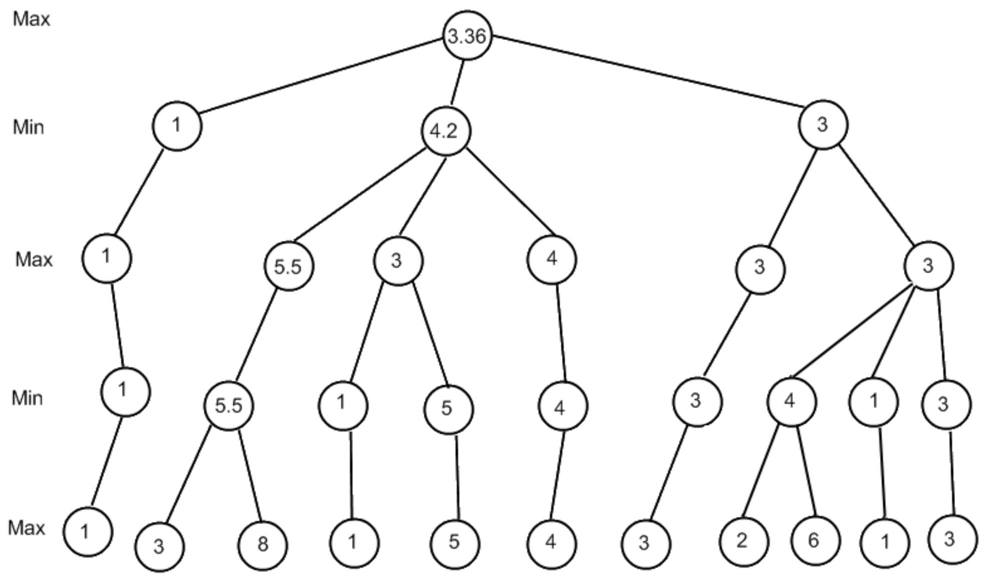

Figure 1: An example MCGTS tree, with the values in each node equal to the weighted average of the values of its children.

Bridge

Bridge is a card game played with four people that are put into pairs, where the members of the same pair sit opposite to each other. Each player is dealt a hand of 13 cards, which are only known to that player. The players take turns making bids until three players in a row pass. The last bid becomes the contract for how many tricks the pair needs to take. Once bidding is complete, the player to the left of the declarer plays the first card. Then the partner of the declarer (the player to make the final bid) lays his hand on the table for all to see. His hand is the “Dummy” hand, and is played by the declarer instead of his partner, who cannot say anything to help. Play proceeds with the player to the left of the declarer, each player taking turns selecting a card to play, going in a clockwise direction. A player must play a card of the same suit as the original card if he has one. After every player has played one card, the winner of that “trick” of four cards is determined based on the cards played. The winning card is the one of highest rank of the same suit as the first card played in the trick, unless a card of the trump suit (can be any suit, or none) is played, then the highest ranking card of the trump suit wins the trick. The winner of the trick starts off the next round of play, and it continues until all cards have been played. The tricks are totaled for each pair, a score is determined, and then the process repeats with a new hand.

Programs that use Monte Carlo methods can be and have been made to play Bridge well. These programs can successfully execute difficult plays that most humans would have trouble discovering, due to their way of selecting moves to make (Rajkumar). It is extremely difficult to determine an optimal move in every situation for Bridge. The best programs play using a Monte Carlo technique because it can provide accurate predictions. To do this, many different sets of player hands are randomly generated, all of which conform to the current knowledge available. This is synonymous with Double Dummy play, where instead of on player’s hand being revealed, there are two Dummy hands, one from each team. This allows the two other players with their hands not revealed to know every player’s hand through deduction. In this program, the MCGTS is used to determine what the best play for each of those hands would be. For Bridge, the MCGTS works by playing out the same hand many times from a starting position in the game. In the tree of moves, the first set of children would be the possible cards the first player could play, and the children of those would be the cards that the second player could play based off of the first player's card, and so on. Once the method has played through an entire hand once, the number of tricks (for the pair that the player of the method is on) is calculated based on the cards played and stored in the bottom child of the tree, which is at the end of the path taken when playing through the hand. Then the value is brought up through the tree, averaged with any other values from previous plays through the hand, using a weighted average based on the number of visits to that node. Eventually this reaches the root of the tree, whose value represents how many tricks the method predicts that the pair will take overall with optimal play for all players. The child of the root with the highest value represents the card that is predicted to be the best to play. Using the knowledge gained with the MCGTS from all of the different hands, the program can predict quite accurately what the best card to play would generally be, as well as roughly how many tricks the pair will take.

This project involves using Monte Carlo methods to approximate the best card to play with a double dummy hand setup. Bridge is a good test bed for this due to its simplicity, and if the method is very fast, it could potentially be applied to actual Bridge programs.

The New Method of Averaging

A component the MCGTS algorithm involves taking a simple weighted mean of values. Except for the bottom leaves of the tree, the value for every node is simply the weighted average of all of its children's values. This average is taken after every time the hand has been played through, as the method goes back up the tree, calculating the new value of each parent node based on the new value of its child, which resulted from playing through the hand. The value of a node is simply the weighted average of its children:

$$X_i = \frac{\sum X_i T_i}{n}$$

Where X is the value of the parent node, X_i is the value of its i^{th} child node, T_i is the number of times the respective child has been visited, and n is the number of times the parent has been visited (Rajkumar).

For this project, a new method of averaging, which from here on will be referred to as the generalized average, is tested to see its effects on the resulting values that the algorithm gives. The general form of the function that is used for generalized averaging is:

$$X_i = \sqrt[p]{\frac{\sum {X_i}^p T_i}{n}}$$

From this equation the similarities to the normal weighted average are apparent. Everything is the same except that the individual values of the children are raised to a certain power, p, before being multiplied by the respective number of visits and summed, and that the result of dividing the sum by the total number of visits is then raised to the power $\frac{1}{p}$, which is the same as taking the p^{th} root of the result. Rivest describes the generalized average in detail, and explains its application to standard game tree searching (Rivest). The involvement of the generalized average in this project is similar to Rivest's application to minimax searching. Unlike the weighted average, which works the same no matter whether it is a partner's turn or an opponent's, the generalized average is affected by this. It needs to be applied to minimax, so that when a partner is playing, we want to emphasize larger values, which the program does, but when an opponent is playing, we want to emphasize the smaller values. This is done by using the opposite values of the values of the children, or in other words, $13 - X_i$, and the final result is the root expression subtracted from 13. The reason for this is that the smaller values become the greater values, allowing them to be emphasized with the same generalized average and still return a small value. The differences in the average that depends on the current player are illustrated in Figure 2. From the figure, it can be seen how the generalized average does have an effect on the value that makes it different from the weighted average just based on the root value. The generalized average is not a single function; it is instead a set of functions, as different powers of p result in different functions. The generalized average with power $p = 1$ is the same thing as the weighted average. The goal of the research is to determine if generalized averaging is an improvement on weighted averaging, and in particular what values of p work the best in different situations, and since the original weighted average is a part of this, it does not have to be tested separately.

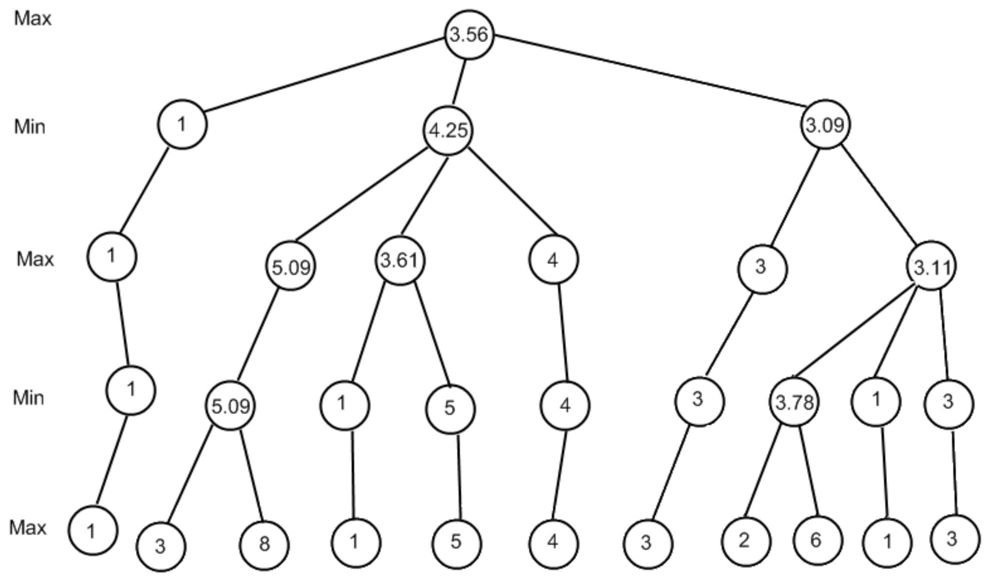

Figure 2: An example of a MCGTS tree, using the generalized average with power of two. The nodes on Min use the opposite way of taking the average, as those are the opponent's moves.

The weighted average that is used in the MCGTS has some flaws that can affect the end result of the method. The major problem with it is the influence that the lower child values can have on the average. The value that is the best is generally the value that you want, and really bad valued children should not be taken into account. While the weighted average does help to take care of this, as the worse valued children have a much lower weight as they are visited less, they still can drag down the better value, especially if they are almost as good as the best. The MCGTS works by balancing between repeatedly searching the best moves and exploring the tree for better ones. It is important for new paths to be explored to try and find more obscure moves that may end up better than the best move found so far. However, the MCGTS needs to continuously search the better moves so that they are weighted enough to cancel out the effects of the explorations into worse moves, otherwise the average would decrease significantly and be less accurate. The benefit of generalized averaging is that even more exploration is possible. What the generalized average does is place greater emphasis on the larger, better values of the children by raising them to a certain power, so the end result is closer to the best values and be less affected by the worse ones, even if the worse moves are still weighted heavily. To have the MCGTS spend more time exploring, all that is needed is to increase the constant *c* in the formula used to determine which move to select, which makes it easier for worse moves to be played. Then the generalized average can be used to dilute the effects of the over-weighted moves that are not good. By combining a high constant with a high power in the generalized average, the algorithm adequately explores the various possible moves yet still results in an accurate value for the best move.

Method

In order to test the benefit of using a higher generalized average instead of just a weighted average, data on many hands needed to be collected. Then, for multiple combinations of constant *c* and power *p*, the error was calculated by comparing the card predicted as the best play from a run of the MCGTS and the actual card that should be played with no trump, as determined by a DDS. The programs used in testing were all written in Java, using the Eclipse

IDE, and the section of the program that used the main algorithm is in the Appendix. This algorithm is run thousands of times, recursively creating the tree one branch at a time. The computer program allows for the selection of a trump suit. It turned out that runs using no trump had empirically higher error, so as a worst case, no trump was used. First, a set of 100 hands were randomly generated using a program to make and record them. For each of those hands, the number of tricks the pair would take for every card in the first player's hand was calculated using the Bridge Captain Double Dummy Solver, which was downloaded for this purpose (Richardson). These values were compiled into a matrix of 100 rows, one for each hand, and 13 columns, one for each card of the first player's hand. Using a program that was created to run the MCGTS for a game of Bridge, the card that was predicted to be the best for each hand for multiple values of the constant and power was determined by running through 5000 plays of every hand. The reason for only playing through the hand 5000 times was that the value for the method easily converged by that point, if not before, so running through only 5000 plays would get a close result and still not take too much time to run. The program was run with values of the constant from 0.125 to 128.0, incrementing by a power of two, and with values of the power from 1 to 15. These values were used to both include the default settings as well as provide a large range of results to work with. For every pair of constant and power, the error for each hand

$$|DDS\ Value\ for\ best\ card - DDS\ Value\ for\ predicted\ best\ card|$$

was calculated, and then the errors for all 100 are averaged together. This represents the average error for that specific combination of constant and power. It represents the average difference between how many tricks the player should take and how many tricks the player would take if the card predicted by the MCGTS program were played instead. Since the DDS is exact and accurate, it was used to determine the actual values, the MCGTS program was used only to find the card it predicted to be the best to play.

Results

Using the data generated from the programs, a chart was created to display the data, depicted in Figure 3. This displays the data where the row represents the value of the constant used, and the column represents the power level used. The colors of the data help make the trend in values of the data easier to visualize. As the chart shows, the lower values tend to reside around constants ranging from 2 to 32, and power levels ranging from 1 to 5. The optimal combination of power and constant are 1 and 16, respectively, having an error of only 0.175, the minimum error of all combinations. The combination of power of 5 and constant of 16 is the next best, with an average error of only 0.18. There also seems to be a trend of low errors around a constant of 1 that moves left along all of the powers. However, the worst errors are generally with a combination of low constants and low powers, or a combination of high constants and high powers.

Average Error		**Power**														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Constant	0.125	0.3	0.34	0.29	0.335	0.28	0.24	0.255	0.265	0.315	0.28	0.255	0.26	0.275	0.265	0.23
	0.25	0.3	0.27	0.28	0.23	0.285	0.275	0.245	0.25	0.255	0.285	0.225	0.27	0.215	0.265	0.245
	0.5	0.2	0.2	0.245	0.21	0.275	0.315	0.225	0.24	0.235	0.265	0.26	0.23	0.22	0.25	0.195
	1	0.25	0.235	0.215	0.2	0.195	0.31	0.295	0.245	0.22	0.245	0.205	0.28	0.275	0.23	0.225
	2	0.25	0.235	0.205	0.24	0.22	0.21	0.22	0.195	0.215	0.235	0.23	0.23	0.22	0.215	0.26
	4	0.255	0.19	0.23	0.225	0.215	0.22	0.215	0.27	0.22	0.235	0.265	0.26	0.25	0.225	0.265
	8	0.21	0.25	0.23	0.22	0.2	0.25	0.25	0.255	0.235	0.31	0.255	0.265	0.28	0.28	0.28
	16	0.175	0.235	0.185	0.215	0.18	0.235	0.22	0.25	0.22	0.255	0.255	0.25	0.265	0.265	0.28
	32	0.195	0.2	0.21	0.205	0.22	0.24	0.225	0.225	0.255	0.275	0.255	0.295	0.265	0.315	0.32
	64	0.235	0.225	0.245	0.24	0.255	0.24	0.275	0.26	0.265	0.32	0.29	0.245	0.335	0.32	0.34
	128	0.29	0.235	0.245	0.27	0.285	0.305	0.3	0.265	0.265	0.33	0.235	0.325	0.28	0.27	0.315

Figure 3: The average |error| for every combination of constant and power that were tested, in number of tricks. The shading helps depict the trend of values of the chart. Green colors mean a lower average error, while red colors represent a greater average error, with yellow shades in between.

Discussion

This project has shown that the use of this new method of averaging, the generalized average, does not improve upon the accuracy of the MCGTS applied to Bridge in determining the best card to play. The best combination of the power and constant, as found by the program, is a power of about 1 and a constant of 16. However, while the MCGTS program may not be better at selecting the right card, it is not at all worse, and with a constant of 16, the difference between using a power level of 1 and a power level of 5 is only 0.05. With more time, additional hands to the 100 could be run to get more accurate results to see if there is more of a difference. The reason behind this may be that the generalized average helps only in terms of how many tricks are predicted to be taken and not in finding the right card to play. With more testing, it is possible that the generalized average could be found to be beneficial in finding the predicted number of tricks using the MCGTS program. If this is the case, then the generalized average could be applied to other games that use the MCGTS to try and improve their accuracy. Another possibility for the future would be to have the power *p* be set dynamically, changing it depending on the sample used.

The MCGTS actually works reasonably well when applied to Bridge, and is quite fast. In the future it could be improved to be even faster by optimizing the code, which would make it more useful. Interestingly, while the Bridge MCGTS program that was created was usually slightly inaccurate by a few tricks, it still made the right plays during the game, which is apparent by the low errors in the results. Out of 100 hands, the average error was usually less than one quarter of a trick. Also, with some modifications, the program could be made more accurate, making it a worthy replacement of a DDS due to its speed, as it runs in a fraction of the time that the DDS takes. However, as it is not a full Bridge program, it would only be a small part of a greater program that just uses it to find the best play for certain hands. Despite its flaws, the program itself was a moderate success at achieving what it had set out to accomplish, even though the power average did not. The program was successful as an artificial intelligence for Bridge, and if it could be run many more than 5000 times and still be efficient and fast, it could potentially improve Bridge programs.

References

Brugmann, B. (1993, October 9). Monte Carlo Go. *Max-Planck-Institute of Physics*. Retrieved from http://www.ideanest.com/vegos/MonteCarloGo.pdf

Gelly, S., & Wang, Y. (Eds.). (n.d.). *Exploration exploitation in Go: UCT for Monte-Carlo Go*. Retrieved from http://www.lri.fr/~gelly/paper/nips_exploration_exploitation_mogo.pdf

Rajkumar, P. (n.d.). *A Survey of Monte-Carlo Techniques in Games*. Retrieved from http://www.cs.umd.edu/Grad/scholarlypapers/papers/Rajkumar.pdf

Richardson, B., & Haglund, B. (2008). Double Dummy Solver (Version 4.2) [Computer program]. Retrieved July 4, 2009, from http://www.bridge-captain.com/downloadDD.html

Rivest, R. L. (1995, March 29). *Game Tree Searching by Min/Max Approximation*. Retrieved from http://people.csail.mit.edu/rivest/Rivest-GameTreeSearchingByMinMaxApproximation.pdf

UCT. (n.d.). Retrieved July 22, 2009, from http://senseis.xmp.net/?UCT

Appendix

Method Code:

```
private static void playRandomGame(int player, ArrayList<ArrayList<Integer>> hands,
ArrayList<Integer> cardsplayed, int thecurrentplayer, DefaultMutableTreeNode root, double
constant, int power){
	boolean maxmin = (player % 2== 0 && thecurrentplayer % 2 == 0) || (player % 2 == 1
&& thecurrentplayer % 2 == 1);
	//^True if player's turn (i.e., max)
	if(cardsplayed.size() == 52){
		int tricks = 0;
		int winner = player;
		for(int i = 0; i < cardsplayed.size() / 4; i++){
			ArrayList<Integer> temptrick = new ArrayList<Integer>();
			for(int j = i * 4; j < i * 4 + 4 && j < cardsplayed.size(); j++){
				temptrick.add(cardsplayed.get(j));
			}
			winner = (getWinner(temptrick) + winner) % 4;
			if(player == winner || player == (winner + 2) % 4)
				tricks++;
		}
		((BridgeNode)root.getUserObject()).setValue(tricks);
		((BridgeNode)root.getUserObject()).visit();
	}
	else{
		DefaultMutableTreeNode next;
		ArrayList<DefaultMutableTreeNode> nodes = new
ArrayList<DefaultMutableTreeNode>();
		for(int i = 0; i < root.getChildCount(); i++){

if(((BridgeNode)((DefaultMutableTreeNode)root.getChildAt(i)).getUserObject()).getVisits() ==
0)
			nodes.add((DefaultMutableTreeNode)root.getChildAt(i));
		}
		if(root.getChildCount() == 0){
			ArrayList<Integer> trick = new ArrayList<Integer>();
			for(int i = 4 * (cardsplayed.size() / 4); i < cardsplayed.size(); i++){
				trick.add(cardsplayed.get(i));
			}
			ArrayList<Integer> playablecards =
Player.getPlayableCards(hands.get(thecurrentplayer), trick);
			for(int card : playablecards){
				root.add(new DefaultMutableTreeNode(new BridgeNode(card,
0)));
			}
			next = (DefaultMutableTreeNode)root.getChildAt((int)(Math.random() *
root.getChildCount()));
		}
		else if(nodes.size() > 0){
			next = nodes.get((int)(Math.random() * nodes.size()));
		}
		else{
			//Select card using UCT Algorithm
				DefaultMutableTreeNode node =
(DefaultMutableTreeNode)root.getChildAt(0);
			double c = constant *
Math.log(((BridgeNode)root.getUserObject()).getVisits());
			double m = ((BridgeNode)node.getUserObject()).getValue();
			double added = Math.sqrt(c /
(((BridgeNode)node.getUserObject()).getVisits()));
			if(maxmin)
				m += added;
			else
				m -= added;
			for(int i = 1; i < root.getChildCount(); i++){
				DefaultMutableTreeNode newnode =
(DefaultMutableTreeNode)root.getChildAt(i);
				double val = ((BridgeNode)newnode.getUserObject()).getValue();
				added = Math.sqrt(c /
(((BridgeNode)newnode.getUserObject()).getVisits()));
				if(maxmin)
					val += added;
				else
					val -= added;
				if(val > m && maxmin || val < m && !maxmin){
					m = val;
					node = newnode;
				}
			}
```

```
				next = node;
			}
				//Run method with child, remove card from hand and add it to
cardsplayed, increment root visits
			int nextplayer = (thecurrentplayer + 1) % 4;
			int card = ((BridgeNode)next.getUserObject()).getCard();
			hands.get(thecurrentplayer).remove((Integer)card);
			cardsplayed.add(card);
			if(cardsplayed.size() % 4 == 0 && cardsplayed.size() != 0){
				ArrayList<Integer> trick = new ArrayList<Integer>();
				for(int i = (cardsplayed.size() / 4 - 1) * 4; i < cardsplayed.size(); i++){
					trick.add(cardsplayed.get(i));
				}
				nextplayer = (getWinner(trick) + thecurrentplayer + 1) % 4;
			}
			playRandomGame(player, hands, cardsplayed, nextplayer, next, constant, power);
			((BridgeNode)root.getUserObject()).visit();
			//Set new value for root
			float mean = 0;
			int count = 0;
			for(int i = 0; i < root.getChildCount(); i++){
				BridgeNode node =
((BridgeNode)((DefaultMutableTreeNode)root.getChildAt(i)).getUserObject());
				if(maxmin)
					mean += Math.pow(node.getValue(), power) * node.getVisits();
				else
					mean += Math.pow(13.0 - node.getValue(), power) *
node.getVisits();
				count += node.getVisits();
			}
			if(maxmin)
				mean = (float)Math.pow(mean / count, 1.0 / power);
			else
				mean = (float)(13.0 - Math.pow(mean / count, 1.0 / power));
			((BridgeNode)root.getUserObject()).setValue(mean);
		}
}
```