Playing Quarto with Monte Carlo Tree Search

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

The ability to plan ahead and strategize has been one of the major facets of artificial intelligence research since its inception. Games have often been used as bench marks to assess the effectiveness, and efficiency of, techniques for planning and strategy. Monte Carlo Tree Search is a method which is currently being used to improve the play of computer programs across a wide variety of games. In this paper we apply the technique of Upper Confidence Trees (one implementation of Monte Carlo Tree Search) to the task of playing the game of Quarto. This was found to be more effective than an algorithm which relied only on information from the next couple of states.

1 Background

Since the beginning of the field of artificial intelligence there has been debate over what defines intelligence. Some of the qualities which people associate with the term include planning, strategy, and pattern recognition. It is these faculties that are used when humans play games, and so creating a program that can play games has been a focus for many computer scientists.

There was a major breakthrough for the field when in 1997 IBM's Deep Blue beat then world champion Garry Kasparov $3\frac{1}{2} - 2\frac{1}{2}$ over a a six game match. This was the first time a computer had beaten the world champion under tournament conditions. However there are several other popular games, which are still played better by top humans than by top machines. One important example of this is the game Go. Go is a much harder game to analyze, since each turn there are hundreds of possible moves, and games take over a hundred moves. By comparison chess games are usually finished in less than a hundred moves, with less than 40 moves available to a player, on average [5]. Note that as a game of chess progresses pieces are removed, and the number of moves to consider rapidly decreases. In Go any unoccupied square is a valid move (with few exceptions), leading to a much less drastic reduction in complexity as the game progresses. in order to deal with this additional complexity inherent in the game of Go new techniques had to be created, which have proven effect for many other games for which tradition tree search had proven intractable.

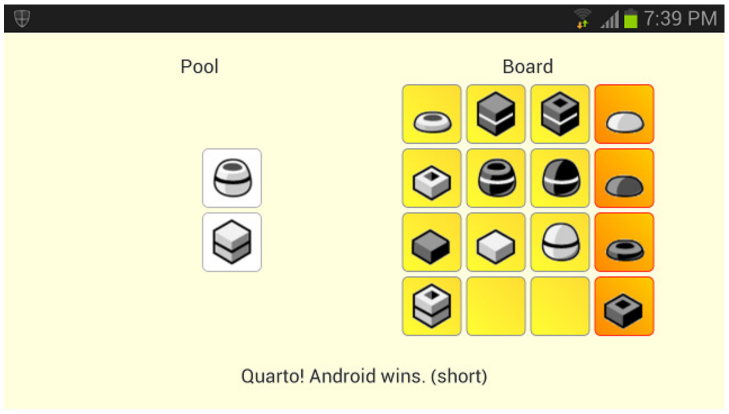
One technique which was introduced around 2006 is Monte Carlo Tree Search (MCTS). Using the power of randomization the skill level of Go playing machines has been brought up to the ability of highly ranked competitors (although not to the level of top players)[6]. Instead of trying to deterministically calculate the best move, one uses random plays to approximate the strength of a given move. This helps avoid wasting time on exploring bad moves, while providing enough information to find the good moves. MTCS has helped improve results for computer programs to play many other games. This includes the game Arimaa, which was specifically designed to be difficult for machines to play [1].

Although the most obvious application of this technique is to other combinatorial board games, it has been used in a wide variety of other applications. Some of these include single-player games, general game players, non-deterministic games, real-time games, optimization, constraint satisfaction, and planning [1][3].

1.1 Quarto

In this paper the technique of MCTS is applied to the game of Quarto. Quarto is a two player board game which was published in 1991. It is played on a four by four grid with a set of sixteen distinct pieces. Each piece corresponds to a specific permutations of four binary features. These features are colour (dark or light), height (tall or short), shape (round or square), and top (flat or indented). The goal of the game is to have produced a line of four pieces which all share a feature. Note that the winner is the player who places the last piece in a row, it is irrelevant who placed the rest of the pieces. Each turn a player places the next piece on the board, and then choose the piece their opponent will play on their turn from the set of pieces that have not been played.

The impartiality of the pieces, and the choice of which piece your opponent will play, lead to much deeper strategy than one might originally expect. In order to win a player must manoeuvre into a position in which the opponent has no choice but to hand them a winning piece.



Quarto! Android wins. (short)

Figure 1: Picture of a free Quarto game for an Android phone.

1.2 Game Theory

In order play a game one must make decisions between a number of possible moves. These moves take the game from it's current state, to some other state. This process can be described in terms of a directed tree. Each node represents the state of the game. The state of can contain many components, in Quarto the important information is where which pieces are on the board, and which piece is to be played next. An edge directed from one node to another indicates that there is a move that takes the game from it's current state to that next state. With most games there will be several terminal states, which correspond to a player winning, or to a draw.

Playing a game can thus be thought of as the task of walking through this tree. Exploring the tree can lead to insight into what next move is ideal. In order to do such a traversal you need to have some way to evaluate the strength of each position in the tree. A simple way to do this is design some kind of heuristic, which evaluates the strength of a given position. The program can then generate all the possible states it can move to from the current state, and evaluate each of them with the heuristic, then choose the best. This technique struggles in that it can be very hard to design effective heuristics.

The only completely effective way to evaluate a position is to consider all the possible moves that could be taken afterwards. It is important to keep in mind that there is an adversary taking half of the moves through this tree. In order to play optimally you must assume your opponent will also play optimally, and thus not offer them any good plays. That is you want the lowest valued response to a move (the optimal state for your opponent to move to), to be as large as possible. The technique that is used to do this kind of search is called minmax, because at each step you try to minimize the maximum move your opponent can make. One assigns values to the terminal nodes based on the result of the game, and then the values of the interior nodes can be recursively calculated [4]. This can be very expensive if the game tree is very large. Instead of starting at the leaves of the tree, heuristics may be used to evaluate the strength of an interior node after expanding the tree to the given depth. This value can then be used in the minmax algorithm to calculate the value of the moves you are choosing between. If the tree has a high branching factor, it may be infeasible to expand the tree to an appropriate depth. Thus more sophisticated techniques are necessary.

2 Monte Carlo Tree Search

Monte Carlo Tree Search is a tree search algorithm to use random plays to approximate the strength of the positions you are deciding between. The specific variant which is being used in this paper is known as Upper Confidence Trees. The main algorithm for determining a single move consists of four stages, a tree search, an expansion of the tree, a ”random” play, and then updating the tree. During this process you store a partial copy of the game tree, which is used throughout the entire game. Each node that has been expanded contains a count of the number of times it has been visited, and the number of times those plays have resulted in wins. Then the strength of a position can be thought of as the win rate of it’s corresponding node.

During the first stage of the algorithm we want to walk from the root of the tree to a node which does not have all it’s children expanded. At each step we go to the most promising child. Note that the win rate at each level corresponds to the win rate of the player who makes that move, so we don’t need to worry about changing our policy for the adversary. When we reach a node which does not have all of it’s children in the tree, it is time for the second stage of the algorithm. A random child of the node, which has not been expanded, is chosen to be added to the tree, and then is made the next move. It is important that this random choice is made uniformly from the set of possible next moves, since any sort of bias in the selection process may bias the score given to its parent.

For the third stage, the game is completed from the node we reached during the first stage using a separate policy. This policy uses randomness to speed up decision making, and doesn’t rely on knowledge about the game, other than recognizing which states represent wins, loses, or ties. In the last stage the path taken my the initial tree search part of the algorithm is retraced, and the information in each node is updated. For each one the visits counter is incremented, the wins counter is updated based on whether or not the player making that move won the game (i.e. for ever other node).

After a certain number of these iterations the program has to decide which child of the root is the strongest. After making its choice the opponent will respond and whatever node they choose will become the new root. Then the entire process starts over, although the tree stores all the results from simulations in previous rounds.

The only choices the programmer has to make is determining the policies for the tree walk, the game play, and which node to move to. Walking through the tree is an optimization problem, we want to find the node which has the highest win rate. This win rate is something we cannot know with out calculating the entire tree, but we can estimate it using the results of our trials as $\frac{w_i}{v_i}$, where w_i is the number of wins in node i and v_i is the number of visits to node i. There is an exploration, exploitation trade off. Nodes that currently have a high win rate are probably good, but those nodes which haven’t been visited as often may have a higher win rate than is evidenced by the simulations completed so far.

There are several different ways to deal with this dilemma, one way to deal with it is to simply make the best move with some probability, and otherwise make a random move, at each step. This is not the most efficient way to deal with the trade-off, and may end causing the algorithm to waste time on moves that have been shown to be quite impractical. A much better approach is to use an upper confidence bound on the win rate as the score to be compared. This is why the technique is called an Upper Confidence Tree. A good upper confidence bound is $\frac{w_i}{v_i} + C\sqrt{\frac{\ln(v_{P(i)})}{v_i}}$ where $P(i)$ is the

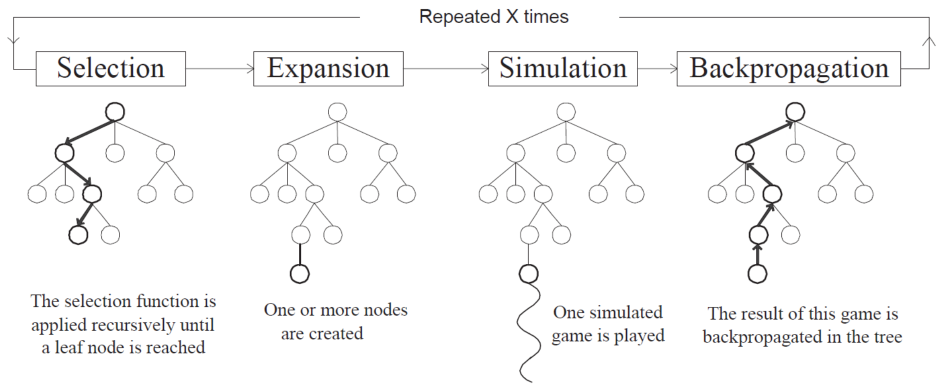

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 2: Diagram of the MCTS algorithm, taken from *Monte-Carlo Tree Search: A New Framework for Game AI* [2]

parent of node i, and C is some constant [2]. The choice of the constant affects how much weight is put on exploration in the algorithm. For large constants you get very even sampling of the children, if the constant is small you focus more on which nodes have the highest win rate. For this paper the exact formula used was $\frac{w_i}{v_i} + \sqrt{\frac{\ln(v_{P(i)})}{5v_i}}$.

For the game policy it is important to keep it as simple as possible, so as to minimize the amount of time the simulation takes, while simultaneously being intelligent enough to provide meaningful data. The first game policy tried moved to a winning position if possible, and otherwise moved randomly. This proved insufficient, as the probability of moving to a losing state proved very high, and thus all nodes had a more or less even win rate. The game policy used was a slight modification. The first step was checking if there's a winning move and placing the piece at random if not, this is the same as before. However, to choose the piece to give to the opponent to play next, a piece was chosen at random from the set of pieces which would not allow the opponent a winning move, instead of from all available pieces. If no such move existed, then the game was recorded as a loss. This policy simulates a player who makes optimal moves, without looking beyond the next move. This allowed the simulations to be played to a much greater depth, and improved the play skill.

The last policy choice that has to be made is which move is selected when it comes to making a decision about which move to actually make, after all the simulations are done. To keep things simple you can simply take the child node which has the largest number of visits. This can be better than taking the node with the highest win rate, since the higher win rate may simply be due to not being explored as thoroughly [2].

3 Results

To test the effectiveness of the UCT approach to playing Quarto it was necessary to test against some opponent. Unfortunately it was not feasible to have it play a highly ranked player, since no such player rankings exist for the game. There was also no easily accessible source code for strong Quarto programs. As a result input and output to such a program would have been done manually. Thus in order to test the strength of the UCT a second Quarto program was written. This program followed the basic pattern of the game policy used to do the simulations for UCT program, with the additional feature that if placing a piece in a certain square led to a losing situation for the player, the other squares are tried. This second program performed well in a few test games against several free online Quarto programs, and the author, hence it was deemed strong enough to test the UCT program.

The test was done by running games between the two programs, half of which the UCT program got to go first, the other half it went second. Four such sets were run, changing the number of

Table 1: Results of Quarto UCT program trials with 10 simulations

	Wins	**Draws**	**Losses**
Going First	304	0	196
Going Second	270	64	166
Total	574	64	362

Table 2: Results of Quarto UCT program trials with 100 simulations

	Wins	**Draws**	**Losses**
Going First	410	0	90
Going Second	321	86	93
Total	731	86	183

simulations run by the UCT program at each step by the UCT program to calculate the top move. The numbers of simulations used were were 10, 100, 1000, and 5000.

The results are summarized in tables 1-4. It is fairly obvious that the UCT program had the advantage with a winrate above $\frac{2}{3}$ in all but the test with only 10 simulations per move. This shows that the information gained from doing random simulations in this manner is useful for determining the best move to make, even when the number of simulations is small. Due to time constraints only 100 trials could be run with the 1000/5000 simulation programs, so it's hard to tell if the decrease in the number of loses when going from 100 simulations to a 1000 or more is significant. These results do show there was some advantage to doing more simulations, and it seems that after 1000 there is not very much advantage to adding more simulations.

There is a very direct trade-off between number of simulations and the processing time necessary to run the program. While the processing time seems to increase linearly with the number of simulations, the strength of the program seems to plateau. This may be due to the fact that as the game length and branching factor are bounded, the game space is finite. If the number of simulations approaches the total number of possible games it would be more efficient to simply solve the game tree. Note that the branching factor for a certain turn in Quarto is the number of available spaces, times the number of available pieces to choose for the next player, which is $(16 - t + 1)(16 - t)$, where t is the turn number. After 11 moves have been made (assuming there was no victor), there are fewer than 2880 possible games left. A number of these will already have been explored by earlier simulations, so there is no need to do 5000 more simulations. However as the time it takes to do simulations also decreases significantly, the benefit to doing fewer simulations may be minimal.

Early on the branching factor is very high, ratio between the number of parent visits and child visits being very large, since each child must be visited once before a child can be visited a second time. This causes the number of visits to each of the roots children is relatively even. Thus each child only has a small number of visits, probably less than its number of children. So it is highly probable that the opponent will choose to move to a node that has not yet been expanded, thus none of the work done for the simulations will carry over. It would save considerable processing time if the first two or three moves were done without having to go through a full set of simulations.

The first few moves could be a approached in a few different ways. Moves could be made completely at random until a certain turn. This would speed up the program significantly, since the first couple of moves are the slowest. A more involved alternative is storing the results of previous games in memory, and using those to build an initial tree. Simulations could then be run as normal, but with a lot more information than could have been generated if previous information hadn't been loaded. When there are many more stored games than simulations played, the opening moves would become stagnated. Therefore it would eventually be necessary to weigh games loaded from memory differently from games played.

Table 3: Results of Quarto UCT program trials with 1000 simulations

	Wins	**Draws**	**Losses**
Going First	44	0	6
Going Second	35	9	6
Total	79	9	12

Table 4: Results of Quarto UCT program trials with 5000 simulations

	Wins	**Draws**	**Losses**
Going First	35	9	6
Going Second	31	13	6
Total	66	22	12

4 Future Work

This section contains questions which were left unanswered by my research.

1. In this paper it was show UCT can be used to create a fairly intelligent Quarto player, but the opponent was not very strong. How would this technique handle more challenging competition?

2. The constant in the upper confidence bound was chosen to strike a balance between exploration and exploitation. How much could optimizing this parameter improve the effectiveness of a UCT Quarto player?

3. Each node generated by the algorithm corresponds a specific sequence of moves in the game tree, however multiple nodes share the same state, if certain moves are made in a different sequence. Compiling the data from these nodes can create a clearer picture of a nodes strength from fewer simulations, however it makes it harder to compute an upper confidence bound if nodes may have multiple parents. Is there an effective way to use redundancies in the tree to improve performance?

4. Similar to the point above, many game have symmetries. Positions, while technically different, may not have any practical differences. For example in Quarto you could replace all the dark pieces with light pieces and vice-versa, and the analysis of the position would be equivalent. You could merge the counts for symmetric states, similar to the way you would merge equivalent states above, however this would be even more computationally expensive. Is there an efficient way to calculate symmetries, and then use them to improve the effectiveness of MCTS algorithms?

References

[1] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[2] G. Chaslot, S.C.J. Bakkes, I. Szita, and P.H.M. Spronck. Monte-Carlo tree search: A new framework for game AI. *Proceedings of the BNAIC 2008, the twentieth Belgian-Dutch Artificial Intelligence Conference*, pages 389–390, 2008.

[3] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] Stuart Russell and Peter Norvig. Artificial intelligence : A modern approach, 2009.

[5] Claude E. Shannon. Xxii. programming a computer for playing chess. *Philosophical Magazine Series 7*, 41(314):256–275, 1950.

[6] Fabien Teytaud and Olivier Teytaud. On the huge benefit of decisive moves in Monte-Carlo Tree Search algorithms. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 359–364. IEEE, August 2010.