DeepGreen – a simple bridge playing program

Introduction

My old bridge coach always used to ask, “What are the three rules of declarer play?”. These rules stuck with me throughout my whole bridge career and I feel that the idea behind them informs the strategy deployed in my code. They were of course: “Count your tricks”, “Count your tricks” and, lastly, “Count your tricks”. Checking, double checking and triple checking is emblematic of what the Monte Carlo Tree Search (MCTS) is all about - checking (through repeated simulation) which card will give the best chance of winning the board. The motivation behind creating this bridge A.I. is twofold; it was a perfect opportunity to reignite my interest in bridge but also a chance to learn about one of the most important reinforcement learning algorithms - furthering broadening my machine learning knowledge.

Aim

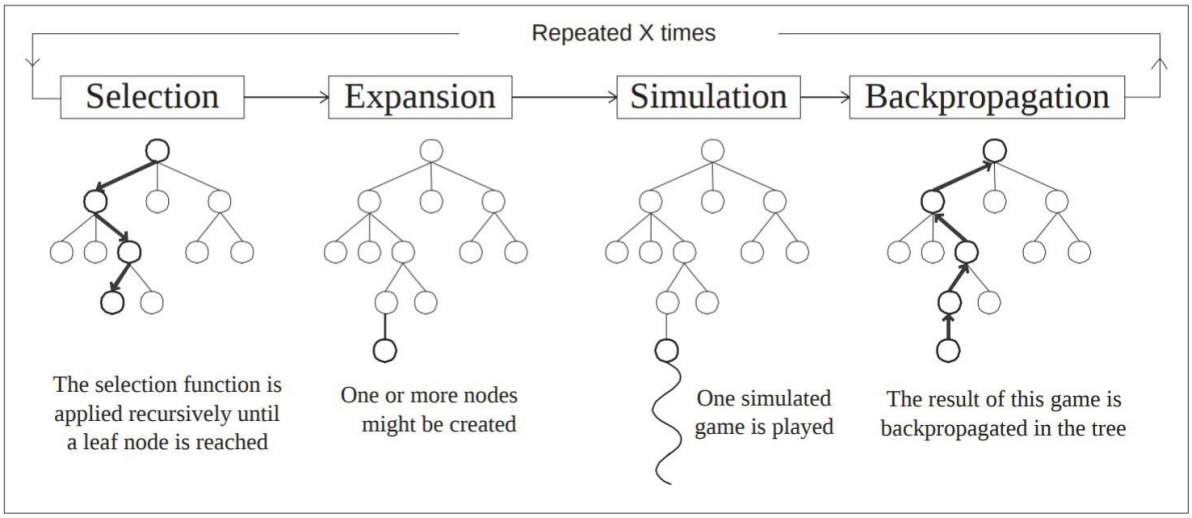
The aim of this project is to develop a double dummy solver which can solve double dummy problems such as those posed in instructional bridge books.

To make it feasible for the project to be completed in time for the deadline the scope was kept limited by developing, not a general bridge A.I. but a double dummy solver. A double dummy problem strips out the two most difficult parts of bridge: the bidding and the uncertainty of distribution of cards in the opponent’s hands. Typically, in a full bridge A.I. the bidding is treated as a separate problem and samples of card distributions are generated based on bidding information and then passed to a double dummy solver which, therefore, is always an important part of a bridge A.I.

For a long time, bridge playing programs were not competitive with advanced players - the search space was simply too vast for the minimax algorithms that had proved effective in Chess. However, in 2001 the GIB algorithm was devised which, using MCTS and various other techniques, was able to compete with expert players. The algorithm was adopted by the largest online bridge website BBO and Monte Carlo methods were employed by many other bridge playing programs.

Monte Carlo Tree Search

Before the Monte Carlo Tree Search minimax was the leading algorithm for turn based game playing programs. Minimax created a tree structure where each node represented the state of the game after making a particular move. Ideally no heuristic is required and the computer can fully expand the tree for the game finding the correct counter to any move made by the opponent. In reality computing power is far too limited and the search space much too vast for this to be feasible for all but the simplest of games. However, the idea of building a game tree out of available moves proved to be a useful one and MCTS develops this idea further to create a statistics tree where the nodes which are expanded (i.e. which moves to make next) are chosen based on how few simulations have been run on that node and how successful past simulations were for that node. MCTS consists of four stages: Selection, Expansion, Simulation and Back Propagation (See Fig.1) which are looped over typically as many times as possible within a given time limit.



Source: [1]

The first stage, Selection, will move down the tree by evaluating every node at its current depth using the upper confidence bound function:

$$UCB = \frac{w_i}{n_i} + \sqrt{\frac{2\ln(N_i)}{n_i}}$$

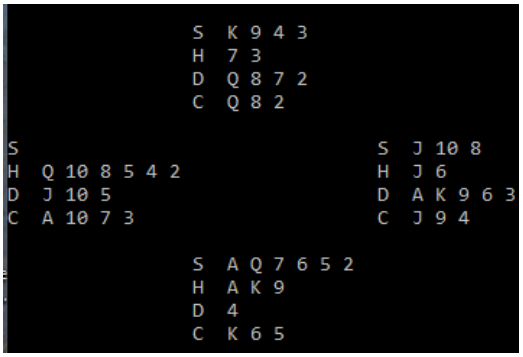
Where n_i is the number of simulations run from that node and w_i the corresponding number of wins in those simulations. N_i is the number of simulations run on the parent of that node. This function is designed to maximise exploration and exploitation where the first term corresponds to exploitation – choosing the nodes with a high win rate – and the second to exploration prompting the algorithm to look at nodes with fewer simulations. The Selection stage ends when the algorithm reaches a terminal node.

The Expansion stage will create child nodes from the terminal node, each corresponding to a valid move that can be made from the game state associated with the terminal node. The Simulation stage then randomly generates random moves from a child node's game state (hence the Monte Carlo namesake) until a winner is decided. The child node is then updated by incrementing its w_i and n_i parameters appropriately and propagating this information back to the root of the tree updating all nodes passed through similarly to the child node. In the case of an adversarial game w_i must be updated according to which player's move a node represents only incrementing if it matches that of the child node thus insuring the optimal opponents moves are also simulated. Following this the rest of the child nodes are simulated and the result backpropagated with the whole process of MCTS being repeated for as many iterations as possible before some condition is met (typically within some time limit).

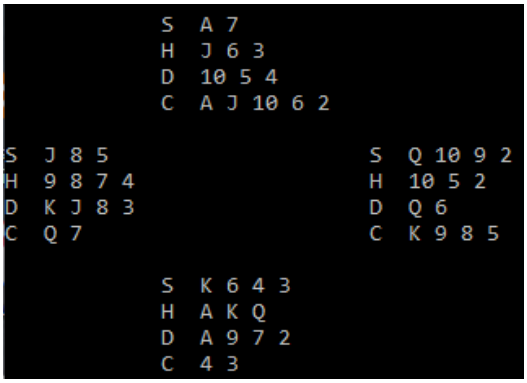
After exiting the MCTS loop the program then looks at all the immediate child nodes coming from the current game state and chooses the node with the largest number of simulations for its next move. Every subsequent time the computer must make its turn it simply runs MCTS again until the game reaches its end.

Method

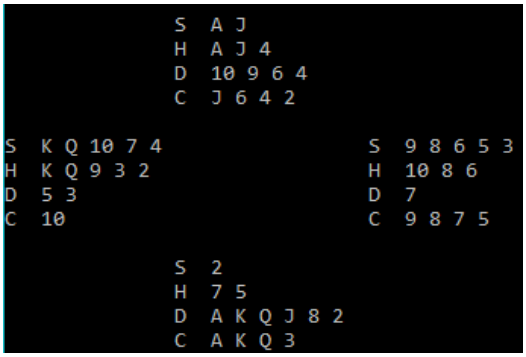
The algorithm was coded such that the computer plays against a human to allow double dummy problems from bridge books to be played with the proper leads and replies to recreate the situations described. I chose three problems to test the algorithm, the first of which is a simple hand where the ruffing is the only extra move required over cashing the trumps and top cards of other suits. The second requires careful play of the club's suit to ensure entries to the dummy are available to cash a third club trick. As this algorithm suffers from the same problem as GIB[2] of putting off tough decisions until later, this is a difficult hand as the entries don't make a difference until close to the end of a game. The third board has 12 clear tricks but requires a squeeze to make the last one which, again, is difficult for this algorithm as it only comes into play in the last few tricks.



Board 1 – 4S by South Source: [3]



Board 2 – 3NT by South Source: [4]



Board 3 – 7D by South Source: [5]

The first board was given (10 * number of valid cards) seconds to run Monte Carlo Search Tree whilst the second was given (20 * number of valid cards) seconds if there were 8 or more valid moves and otherwise (5 * number of valid cards) seconds. Finally, the third board was given 5 minutes if there were more than 8 cards and (10 * number of valid cards) seconds otherwise. These times were chosen somewhat empirically in order to reflect the relative difficulty of the boards. It is difficult to come up with a more general scheme for allocating time spent running MCTS as it requires significant knowledge bridge to decide which moves are difficult and, particularly, tough for the MCTS algorithm to find.

Results

Initially the program struggled with first board when only given 30 seconds to run MCTS but this proved to be very inconsistent with the algorithm failing to make the contract in most cases. As such, the scheme described in the method section was implemented instead resulting in more consistent wins. However, it was found that somewhat similar strategies were chosen in the early stages of the game both before and after increasing the computation time – the differences tended to appear later in the board. These early strategies would tend to differ from the conventional way a human player would play. As the algorithm is only concerned with winning this one board (and indeed, is solving a double dummy problem) it chose strategies that would not be used in conventional play. In this board, an example of that was pulling trumps through a heart ruff instead of just playing spades. Despite its strange tactics the algorithm was still able to win this board although it clearly takes too long to be viable as an opponent in a game.

The second board required a further increase in computing time – likely for the reasons mentioned in the method section. The point of this board was that the first round of clubs had to be ducked in dummy otherwise East could later prevent the declarer from making enough tricks in clubs. The algorithm managed to find this and win the game. Before increasing the computation time, the algorithm would typically act greedy and win the first round of clubs. This is perhaps unsurprising as winning the first round gives a sure trick which increases the likelihood of random simulations winning the overall game. By increasing the computation time, more of the opponents moves are chosen from the statistics tree rather than randomly meaning that taking quick tricks is more likely to be countered later in the game with intelligent replies.

MCTS's inability to look far into the future is highlighted in the third board. Whereas a human could very easily see the 12 tricks and then play cards tactically to take advantage of the discards made from East and West, the algorithm can't play so intelligently. In order to find the right order to play the winners in such that the squeeze can be performed at the end, the algorithm would have to grow the statistics tree incredibly large so that it has a lot of data on the endgame scenarios. This was not feasible for the current instantiation of the program which played the cards required for the squeeze quite early on.

Discussion

The three case studies clearly indicate that using MCTS alone is not sufficient for a functional bridge A.I. as the computation time is unreasonable for even the simplest of hands. The results do however show the potential of using MCTS in tandem with knowledge of tactics and other simplifications could prove fruitful. By modifying the code such that touching cards are not considered separate possible moves the algorithm was able to run significantly faster – improving on boards 1 and 2. Further modifications will have to be made for the algorithm to be able to win the third board as although this modification showed promise – in that it didn't try to win the jack of hearts or play spades in the early tricks – it did succumb to those desperate strategies eventually.

To reduce the algorithm's search space, it would be wise to implement more guidance in the way of tactics. As human players always begin by counting their tricks, any node of the statistics tree where one of those sure tricks from the beginning is lost should be immediately discounted by giving it a very large negative winCNT value, ensuring it is never selected again.

The converse could also be carried out where tactics generally used by humans can be recognised and winCNT increased by slightly more than one in order to increase the chance of that avenue being explored. It no doubt would have helped in board 1 to focus on potential finesses as they are a simple way to win the contract while being able to detect a squeeze and promote exploration of the branches from its node in the statistics tree would likely allow the algorithm to win the third board.

Conclusion

Monte Carlo Tree Search serves as a good basis from which to build a competent bridge playing program. However, the algorithm is severely limited without game specific knowledge as it will allocate computing time to exploring paths with low eventual yields.

References

[1] Iolis, Boris & Bontempi, Gianluca. (2010). Comparison of Selection Strategies in Monte-Carlo Tree Search for Computer Poker.

[2] Matthew L. Ginsberg (2001). GIB: Imperfect Information in a Computationally Challenging Game.

[3] Hugh Kelsey (1979). Winning Card Play. Pg57

[4] Hugh Kelsey (1979). Winning Card Play. Pg88

[5] Charles H. Goren (1985). Goren's New Bridge Complete. Pg602