Poker Agent with Monte Carlo Tree Search

Ong Kuan Yang, Rahul Baid, Ruhani Suri, Soh Boon Jun
Team no. 27
National University of Singapore
A0158348U, A0176876H, A0177443Y, A0168082B
kuanyang.ong,rahul.baid,suriruhani,boonjun@u.nus.edu.sg

Abstract

Poker is a partially observable, stochastic and a imperfect information game and this poses challenges to the traditional search algorithms. In this paper, we will discuss the design of a competent Artificially Intelligent (AI) agent designed to play and win at a specific version of the Poker game. *Taurus* is an AI agent designed to play the game of a 2-player (also known as Heads-up) Limit Texas Hold'em Poker. It uses the **Monte Carlo Tree Search [MCTS]** algorithm integrated with **Opponent Modelling** and **Neural Networks**.

1 Introduction

With partial knowledge, risk and stakes management, unreliable information and even deception, poker has proved to be a tough game for computers to model and conquer. In this paper, we detail our attempt at designing *Taurus*, an AI Poker Agent which aims to be as close to an optimal bot as possible.

Attempting to `solve` such a problem will give us a deeper insight into how computers can be fundamentally trained to imitate human thought process, thus allowing machines to function in an intelligent manner and exhibit intellectual behaviour and learning processes.

We will begin our discussion by briefly describing the game's characteristics and rules [The Game, Section 1.1]. We will then discuss the rationale behind the choice of the MCTS algorithm and how we optimize the different stages of its working [Implementation of MCTS, Section 1.2]. We will examine the benefits of using Opponent Modelling in our implementing to determine a probabilistic distribution of the opponent's hidden cards [Integrating Opponent Modelling with MCTS, Section 3] and how we can make use of Neural Networks to guess the action of a player by observing the environment so we are able to choose a suitable node to expand [Neural Networks to Study Node Expansion Optimization, Appendix A, Section 5.1] . We will thereon analyse the results [Results and Conclusion, Section 5] we obtained and how they improved over the course of our implementation.

1.1 The Game

Poker is a partially observable, extensive form and zero-sum game with uncertainty. It also has a multi-agent environment, which in our case is restricted to 2 players. Lastly, it is also stochastic, as cards are drawn in a random manner. In this paper, we talk about a specific type of poker game, called **Heads-Up Limit Texas Hold'em Poker**. It is a 2 player game with four betting rounds and an initial pot size of $10,000. The small blind begins by betting $10 and the big blind bets $20 at the beginning of the *pre-flop* round. Players have the option to *check, bet, raise* or *fold* after each deal. Cards are revealed in order of 3-1-1 in the subsequent *flop, turn* and *river* rounds until the final *showdown*. Thus, a player wins money and consequently, the game, only if they either have the best hand combination of cards or they are the only active betting player [ie. the other folds].

The fact that we play against only one opponent allows us to implement a possible solution using adversarial search strategy, that is by solving a *minimax* game in which all the possible moves of the players are captured. However, it may be observed that such a computation of all possible moves as a form of the nodes of the minimax tree will result in a vast search-tree, plausibly one which is too exhaustive to traverse. This is where we thought it was necessary to use a strategic node selection and expansion technique, such as the one employed by the Monte Carlo Tree Search [MCTS] Algorithm.

Significant advantages of using MCTS over a general minimax algorithm is as follows:

1. Aheuristic algorithm: Minimax needs a heuristic function to determine the strength of the game state. For games like Poker for which this is difficult to obtain, *aheuristic* algorithms like MCTS come in handy.
2. Search-Asymmetric algorithm: Despite having a high branching factor, MCTS selects and expands nodes which are `good` as decided by the information it learns from its previous moves. The longer the algorithm is allowed to run, the closer to optimal solution it will be able to obtain.
3. Anytime algorithm: Minimax would need to run till the very end to be able to output a solution, which places a non-negotiable bound on its space and time constraints. This is particularly unsuitable for games with large state spaces, such as Poker. MCTS, on the other hand, does not need to run to completion and will always give a reasonable solution when terminated at any time. The longer it is allowed to run, the better solutions it will

output.

MCTS converges to Nash-Equilibrium and has been shown to produce good result in perfect-information games such as Go. Even if used without Opponent Modelling, MCTS is a powerful technique which can easily outperform simple rule-based agents.

There are two main constraints, specific to our project, that we face in the design of our poker agent. Firstly, our poker agent has to respond with an action in the short time limit of *200ms*. Secondly, our poker agent is limited to a file size of *50MB* (though this was later extended).

There are two main approaches to the design of a poker AI. The first involves the pre-computation of a complete strategy that approximates a Nash-Equilibrium. In fact, Heads-up Limit Texas Hold-em has been weakly solved and an AI, *Cepheus*, that is practically unbeatable, has been created using the technique known as **Counterfactual Regret Minimisation (CFR)**.

The second approach is to play the game, similar to how a human would, by thinking in terms of the future game states that are possible from the current situation and trying to optimize the expected winnings. This is known as a **search-based approach**.

We chose the second approach for three reasons. Firstly, our project specifications does not give us enough memory to store a complete strategy. Secondly, while the first strategy is guaranteed to not lose against any player, it is not the case that it is designed to maximise the winnings against an imperfect player. In real life, most players are imperfect and thus the second strategy will allow us to take advantage of weaknesses in the player to maximise our winnings, a technique known as opponent modelling. Lastly, we study an attempt at integrating artificial Neural Networks to our agent to optimize how the Node Expansion decision are made, and why it was not incorporated in the final implementation at the end.

2 Implementation of Monte Carlo Tree Search

Monte Carlo Tree Search algorithm is a aheuristic search algorithm notably employed in game-play decision processes and was introduced by Rémi Coulom in 2006. It is designed by combining the tree search algorithm with Monte-Carlo evaluation and was first used in the implementation of Crazy Stone.[1] It combines the generality of random simulations with the precision of tree search and has four major phases: *Selection, Expansion, Simulation* and *Backpropagation*.

In the basic version of MCTS, the involvement of a heuristic is not required due to the generality of the random simulations by which random actions are taken until a terminal state is reached, whose value is then returned and tracked. However, these random simulations are progressively improved by taking into accounts the previously played games. This is advantageous especially in a complex game like Poker where the heuristic function might not be as easy to obtain. However, this means that this algorithm does not always return the most accurate value, and this is especially so in a time-limited environment such as ours.

To describe the algorithm in a nutshell, we start by picking moves randomly, the result of each of these games is back-propagated to update our beliefs and next time we make a more informed decision when choosing an action. Assuming the opponent plays optimally [ie. if implemented without Opponent Modelling], the MCTS traverses sub-graphs of the tree with the best moves and increases the depth at which it is able to iterate.

Before we begin discussing our MCTS algorithm, we have to define the nodes of the tree. We will define an action node as one where the agent decides which of the 4 actions of *fold, check, raise* and *see* it will perform. In addition, we have chance nodes to represent the random drawing of cards. Lastly for nodes that represent an end state (e.g. showdown or one person folds), we will define the value of a game state as the money made in profit at that state.

The four main stages of MCTS as implemented by *Taurus* are described below. These stages are repeated until infinity, till the algorithm is explicitly stopped, or in our case, when we reach the time limit.

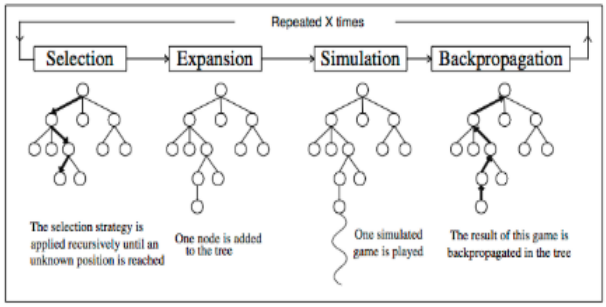

Figure 1: *Pictorial representation of the MCTS algorithm, diagram from Chaslot (2006)*

1. Selection: Starting from the root node, the selection phase entails repeatedly selecting a child node to explore and then exploring it [as described by phase 2 below]. Here, we would like to achieve a smart balance between exploration and exploitation, thus no simple algorithm will suffice as a selection function. For instance, choosing nodes randomly [a random selection function] will perform well on exploration but badly on exploitation. To strike an optimal balance between the two, we will use a type of bandit algorithm known as Upper Confidence Bound (UCB) algorithm [discussed in detail in Upper Confidence Bound Algorithm for Optimized Node Selection, Section 2.1].
2. Expansion: By selecting one child of the current node in Step 1, we have added a new set of nodes into the tree which are open to selection and exploration, and are unvisited currently. Now, we iteratively expand the tree by choosing one of these. At each simulation, the tree depth is incremented by a unit [multiple nodes can also be expanded per simulation but it becomes highly memory inefficient, especially for games with high branching

[1] A 9x9 Go playing program that won the 10th KGS computer-Go tournament.

factor like Poker].

3. Simulation: This phase is where the main work of the algorithm happens. We keep 'simulating' the selection and expansion of nodes from here one to see what final game state, ie. terminal node, it arrives at. We observe and note the winner reached by these simulations. No new nodes are created and no part of the process is stored. The strength of this terminal node is determined by measuring the money profited when all the cards are dealt out. Each MCTS iteration executes this process twice, once from the perspective of each of the 2 players, with different knowledge of the hole cards.
4. Back-propagation: This phase is what makes the MCTS an 'intelligent' algorithm over the others. Once we reach a terminal node in the simulation phase above, we back track our path and update each node that we visited to reach here. We updated the value of the node, or additionally, the number of times it was visited. Now, whenever the algorithm times out, a selection strategy is used to select the best moves based on the data we have maintained about the different paths we saw. An example of a decent selection technique is selecting the node with the highest value [as defined earlier].

2.1 Upper Confidence Bound Algorithm for Optimized Node Selection

As explained in the previous section, we will be utilizing the UCB Algorithm during the selection phase of the process to balance between exploration and exploitation. More specifically, we will employ the following variant of UCB to help us select and traverse nodes. The formula for the UCB algorithm is as follows.

$$E_i + c\sqrt{\frac{lns_p}{s_i}} \tag{1}$$

E_i stands for the expected winnings at node i, which will be determined as more simulations are done at any of the child nodes of node i. s_i stands for the total number of simulations done at node i and s_p stands for the total number of simulation done at node i's parent. Altogether, we can view E_i as the exploitation term and $\sqrt{ln(s_p)/s_i}$ as the exploration term. c is the coefficient that determines how much exploration and exploitation should be done by deciding what weight is placed on the exploration term. The higher c is, the more we will be favouring exploration over exploitation.

Having a good balance of exploitation and exploration is essential and choosing a good value of c is vital for the MCTS algorithm to perform well. However, what is considered as `good` is not very clear since such a complex modelling is affected by multiple factors in the environment (such as the small blind amount, and the range of the expected winnings). Thus, we will be choosing c via an empirical approach.

The basic idea behind our approach to choosing c is to pit different MCTS poker agents with different values of c against each other. In our experiment, the MCTS poker agents will be playing a total of 100 games against each other. The games will be following the exact same rules as that of the competition framework, whereby each agent will start off with a stack size of $10,000. In each game, there will be a maximum of 500 rounds and the agents will have to come up with an action within a time limit of 200ms. An agent will be deemed the winner if it ended off with a larger stack size than what it started with.

The agent with the highest win rate after playing with all other agents will be assumed to have a more optimal c value. In our experiment, we started off by performing basic brute force through multiple different values of c against a Baseline MCTS poker agent with an arbitrarily chosen value of 100.

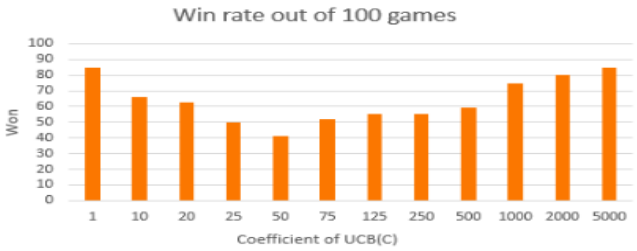

Figure 2: *Win rate of Baseline* $c = 100$ *against MCTS Poker Agents with c between 1 and 5000*

From Figure 2 above, we can see that poker agents with a large $c > 1000$ do not fare well against the Baseline. A low c is also not ideal as from the diagram itself we can see that the Baseline Agent won 85 out of 100 games against another MCTS Agent with $c = 1$. The most promising MCTS Poker agent seems to be having a value of c around that of the 50 [as observed by the shortest bar in the graph above]. Therefore, we will next be performing a second round of experiments with multiple MCTS Poker agents against the MCTS poker agent with $c = 50$ as the Baseline.

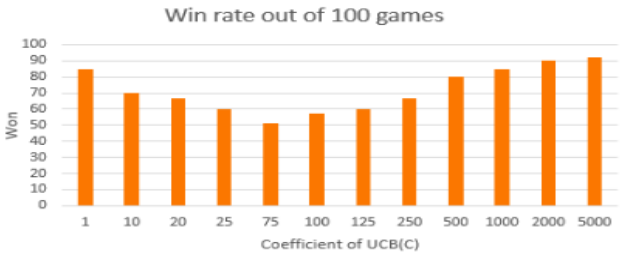

Figure 3: *Win rate of Baseline* $c = 50$ *against MCTS Poker Agents with c between 1 and 5000*

From Figure 3, we can see that an MCTS Poker Agent with $c = 50$ is generally able to perform well against most of the other MCTS agents, over the entire rage of possible c values. However, from Figure 4, we can also see that an MCTS Agent with $c = 50$ does not outperform other Agents with c values ranging from 30 - 100 strongly. In fact, in most games that an MCTS agent with $c = 50$ plays, it only edges out the other agents with c values ranging from 10 - 100 slightly, winning less than $1000 after 500 rounds. However, we will still settle with a c value of 50 since the performance of agents with c ranging from 30-100 is rather similar and it is still able to outperform agents with high $c > 250$ and low levels $c < 20$.

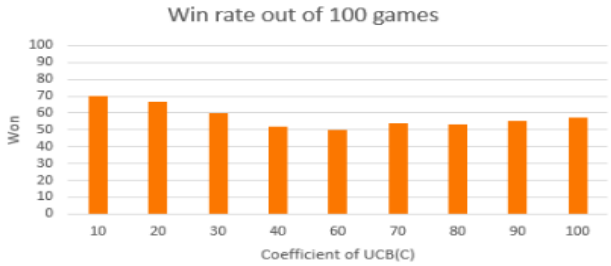

Figure 4: *Win rate of Baseline $c = 50$ against MCTS Poker Agents with c between 10 and 100*

3 Integrating Opponent Modelling with MCTS

Our vanilla MCTS poker AI is moderately strong and is able to win basic AI agents such as players that always raise, check or select a random action in a game of 500 rounds. However, pitting against stronger AIs, such as those found in the class tournament, we found that we were losing by a large margin. This can be attributed to two reasons. Firstly, MCTS does not take into account the opponent's past actions. In reality, past actions are often a good tell-tale sign of how good the opponent's cards are (e.g. raising is a sign of having good cards). This attribute is closer to how a human being would play the game. Secondly, MCTS is too honest. If it has good cards it will raise, call if has average cards and fold for bad cards. This makes it easily exploitable by AIs that recognize this pattern, e.g. an opponent can fold if it sees a raise because it knows its cards are probably worse. This renders our MCTS agent susceptible to being exploited in the long run as it cannot do things like bluffing (raising when it has a bad card to force a fold) or calling when it has a good card to lead the opponent on.

This can be solved by opponent modelling. The basic idea behind opponent modelling is that we use past rounds' data with the same opponent to learn the opponent's playing patterns and account for it when making our decision in subsequent rounds. It be used to predict the opponent's cards in the current round given his past actions. In addition, given a guess of his cards, we can predict his future responses to our actions. These two predictions complement the MCTS algorithms well. The card prediction can be used in the simulation step to get a more plausible expected winning. Secondly, the opponent action prediction can be used in the selection stage to select nodes that we predict the opponent is likely to take given his possible hole cards.

Modern techniques of opponent modelling tend to be quite sophisticated, involving Bayesian statistics and even deep learning. However, these techniques require a long run time (which we do not have) as well as a long history of play with the opponent to collect data (which also we do not have). Thus, we came up with our own simple technique to do opponent modelling. We keep a tally of opponent's actions given opponent's card, previous action by the other player (us) and pot amount. This gives us a probability distribution to predict action given the cards, previous action and pot amount. In addition, we keep a tally of opponent's cards given opponent's action, previous action by us, and pot amount. This gives us the probability distribution to predict opponent's cards. Since there are so many possible cards, and possible pot amounts, it is infeasible to predict exact cards given an exact pot amount given the limited data that we have. Thus, in our tally, we discretize the pot amount into four levels, and we split the cards into buckets of differing hand strengths based on the current community cards.

A problem with our approach is that it only works when showdown occurs because otherwise, the hole cards are not revealed. To solve this, we make use of a heuristic that if opponent folds, his cards are likely to be bad, and if we fold, opponent's cards are likely to be good.

With opponent modelling, our poker AI theoretically gains the ability to adopt human-like strategies such as bluffing. It will think as follows: Given a raise, opponent is likely to fold, hence I will raise even if my cards are bad. Having implemented this opponent modelling, our AI is able to beat vanilla MCTS convincingly, exhausting all its money $20000 in the course of 300 rounds. It has also achieved good results in the class tournament, beating the defending champion (though the tournament was cancelled midway so results are unknown).

4 Results and Conclusion

Taurus has been implemented using MCTS with Opponent Modelling. This entails having to select, expand, and simulate the game tree at different nodes starting from the node. Later, the result of the simulation is back-tracked along the path and included in the maintained states of all the traversed nodes. This allows us to update our beliefs as we choose nodes to explore in subsequent rounds. To further improve our selection functions, we incorporated the Bandit Algorithm called Upper Confidence Bound (UCB) Algorithm to aid in the optimized selection of nodes in Selection Phase [Phase 2] of the MCTS algorithm. For the coefficient selection of the UCB formula, we run experiments by pitting agents with different values of c against each other and observe how they fair, and thus empirically conclude to take the cvalue as 50. We then incorporate Opponent Modelling into our implementation to keep track of the opposition's actions and use this to guess what cards they may have as well as be able to bluff it's actions to confuse the opposition. Lastly, we demonstrate how Neural Networks can be used to improve the selection function at the Expansion Phase [Phase 3] of the MCTS algorithm by keeping track of various features of the environment. When put against different opponents for a 100 rounds per game, we observe the following statistics:

1. A 60% win rate against an agent implemented with MCTS without Opponent Modelling
2. A 65% win rate against a player that raises all the time
3. A 100% win rate against a random player

We hypothesize that these results can be improved further if our agent is given more time to learn.

5 Appendix A

5.1 Neural Networks to study Node Expansion optimization

The intuition for selecting a good node for exploration is that exploring certain nodes might be more favourable than the other and will provide valuable information for the agent to make a more rational decision in subsequent rounds. In the Selection Phase [Phase 2], we have the UCB (see Upper Confidence Bound Algorithm for Optimized Node Selection, section 2.1) algorithm to balance between exploration and exploitation. However, no such selection function is used in the Expansion Phase [Phase 3]. The basic implementation of MCTS currently does not have an algorithm to decide which node is suitable for expansion and a random node is simply chosen at random at this stage. Therefore, in this section, we will be discussing a possible attempt to utilize Deep Learning with Neural Networks to study and determine whether a more optimal node can be estimated for expansion and hence provide a performance boost.

To help us observe if there exists a set of features that can help us choose a more optimal node, we will be utilizing an artificial Neural Network to guess the action of the player given a set of features in the environment that results in him winning the game. An intuitive reasoning for this choice is that the player might have chosen a node to expand more often and hence the agent victory can be attributed to those choices of the nodes.

More formally, we are trying to let the neural network estimate the following formula

$$P(actions\_expanded|Features \wedge agentwon) \quad (2)$$

Our inputs to the neural network will be a set of features in the environment and the output will the the action that the player has chosen.

We first gathered 320,000 sets of playing data from self-play, playing against a random agent and playing against an agent that always raises. 300,000 sets of playing data would be used for training the Neural Network and the other 20,000 will be used for testing. Each set of playing data is a (features, action) tuple.

The following are the set of features that we have considered:

Features — Description
Hand Potential — Winning rate estimated from 500 Monte Carlo Simulations
Max/Min node — -
Actions Available — Raise/Call/Fold
Hand Strength — Estimated Hand strength with Chen's Formula
Pot size — Amount of money both players are risking for the current round
Opponent's Actions — History of actions (Raise/Call/Fold) that the opponent has executed

After training the Neural Network, the following are some of our results: Other than the above mentioned features, we also tried to train Neural Networks with features including Pot size, and opponent's past actions. However, the results from training the Neural Networks are mostly similar with no visible improvements in prediction accuracy across epochs, and an agent that chooses a node randomly for expansion will most likely be able to perform as well as an agent that considers multiple features when choosing a node to expand. However, a possible problem with our approach is that we only considered playing data whereby our agent has won. Parameters of the Neural Network might also have been tuned wrongly, hence explaining the lack of positive results, thus we choose to remove this incorporation from our final implementation.

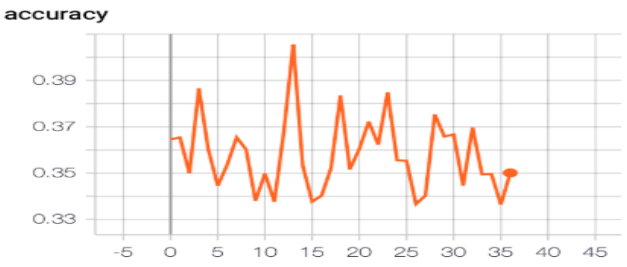

Figure 5: *Prediction Accuracy with Features: [Hand Potential, Max/Min Node, Actions available, Street]*
Potential of a hand is a value from 0 to 1 and it is estimated by performing 500 Monte Carlo simulations with the set of known community cards. The higher the Potential, the more games it won in the simulations.

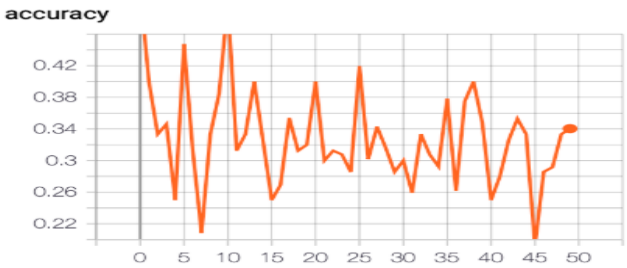

Figure 6: *Accuracy with Features: [Chen Formula , Max/Min Node, Actions available, Street]*

Figure 7: *Prediction accuracy with features: [Hand Potential, Chen Formula , Max/Min Node, Actions available, Street]*

6 References

https://arxiv.org/pdf/1509.06731.pdf
https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa
http://mcts.ai/about/

https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
https://www.aaai.org/Papers/AAAI/1998/AAAI98-070.pdf
https://www.aaai.org/ocs/index.php/WS/AAAIW10/paper/viewFile/1984/2462
[MCTS Fig1]
http://www.scitepress.org/Papers/2014/51489/51489.pdf
http://www.thepokerbank.com/strategy/basic/starting-hand-selection/chen-formula/
https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238