Applying the Monte Carlo Tree Search Algorithm to the Game of Schnapsen

Allison Fleming, Irina Shcherbakova, Ruixin Tang

January 2019

1 Abstract

Computer programs based on the Monte Carlo Tree Search algorithm have achieved significant success in playing against human professionals in the complex Chinese game of Go. Due to its high level of performance, this algorithm was also later successfully applied in other domains. This paper describes the principle of the Monte Carlo Tree Search method and investigates the effect of applying it in the card game of Schnapsen.

The bot guided by the Monte Carlo Tree Search algorithm was created and played against other bots which were guided by different strategies. The results obtained after different experiments with these bots demonstrate that the bot using Monte Carlo Tree Search outperforms all other bots with respect to the number of victories. Moreover, the performance of random and Upper Confidence Bound selection policies of the Monte Carlo Tree Search algorithm was compared and it was concluded that the former shows much stronger results.

2 Introduction

Computer games represent a challenging and fascinating field of research for Artificial Intelligence. They require decisions to be made sequentially with the aim to achieve the maximum benefit or to win a game in the long prospective. In the 1990s, the prevalent strategies used by game programs involved brute-force tree search and alpha-beta pruning [6]. Specifically, the alpha-beta method was employed in the Deep Blue computer which famously won a chess game against the world champion Garry Kasparov in 1996 [6]. However, this method was not sufficient to win in the ancient Chinese game of Go. Go was challenging for Artificial Intelligence algorithms due to its very high branching factor, on the order of around two hundred possible moves for each player [2]. Moreover, the game process is normally developed over a long period of time. The number of total moves each player must make can reach as high as three hundred, making the search tree quite deep [2]. In addition, players have to be very careful in making their moves because each move has the ability to influence the player's

current state and future states until the end of the game [2]. Due to the far reaching impact of each move, it is very problematic to apply a heuristic in Go that can fairly evaluate a board position and state of the game [2].

Defeating the game of Go became a more doable task due to the formulation of a new algorithm. In particular, in 2006 Coulom designed the principle of Monte Carlo Tree Search (MCTS) by incorporating the Monte Carlo algorithm and the process of building a search tree cumulatively [10]. Also in 2006, MCTS was strengthened with the technique called the Upper Confidence Bounds (UCB) which was formulated by Kocsis and Szepesvari [4, 10]. This algorithm showed considerable success in allowing a computer to play Go competitively with professional human players. A computer version of the game Go implementing UCB improved its performance from a 4 dan amateur level to a 9 dan level by winning against the top professional player of Go, Zhou Junxun, in 2009 [2]. Later the MCTS algorithm was successfully applied not only in other games such as Hex, Havannah, Lines of Action, but also in other fields of programming, for instance, energy optimization problems, domain independent planning, solving Markov decision processes, and other areas [6].

This paper describes the effect of applying the Monte Carlo Tree Search (MCTS) strategy to the game of Schnapsen. In this research we are implementing the Monte Carlo Tree Search algorithm in a bot, the MCTS bot, in order to competitively play a card game called Schnapsen against other bots. The main opponents of the MCTS bot are a rule-based bot, mybot, and a Monte Carlo sampling bot, rdeep bot. Our version of the MCTS bot implements MCTS using the Upper Confidence Bounds methods in the selection phase of the algorithm. The main hypothesis which we test in our research is that the bot using the MCTS algorithm can defeat other bots in a majority of games, namely in at least seventy percent of the cases.

The structure of this paper is as follows. Section 3 of the paper provides an overview of the Schnapsen game, its rules, and phases. It also introduces the MCTS algorithm and its most essential method - Upper Confidence Bounds. Section 4 investigates in detail the research question of the paper and our expectations about the performance of the MCTS bot. In section 5 experimental setup of the research is presented, in particular, the description of all bots and the strategies they use. Section 6 shows the obtained results of the experiment. Firstly, it reveals the strongest opponent of the MCTS bot and then gives the outcome of playing the MCTS bot against all other default bots. Secondly, it demonstrates the results of testing the MCTS bot against machine-learning versions of all other bots and the influence of iteration and selection policy on the performance of the MCTS algorithm. Section 7 gives our interpretation of the obtained results and outlines the most important findings of the research. Section 8 presents conclusion of the experiment and summarizes the most important aspects of the research paper.

3 Background Information

For this project we have used an Intelligent System framework provided by an instructor [7].

3.1 Schnapsen

Schnapsen[8] is a two-player card game especially popular in Austria and the southern part of Germany. The game is played as a series of deals and in each deal 0-3 game points can be awarded. The winner of the game has to collect at least 7 game points. Each deal consists of a series of tricks. The player who collects 66 trick points wins the deal. The deck in Schnapsen consists of 20 cards and each card has a rank. A Jack's rank is 2 points, Queen - 3 points, King - 4 points, Ten - 10, and Ace - 11. In the beginning of the deal, each player receives 5 cards. The remaining cards are placed face down in the stock with the last card open as a trump. A trump card can beat any non-trump card or a trump card which is lower in rank. Players collect trick points by winning tricks and declaring marriages. A player can declare a marriage if he/she has both King and Queen of the same suit in the hand. [8]

The Schnapsen game can be divided into two phases. The first phase of the game is characterized as an imperfect information game in which players do not know what cards the opponent has in their hand. This results in a high branching factor of the possible moves of the opponent, eg. in the first trick the opponent can play one of 14 unknown cards. In the second phase, the stock is exhausted. Players know which cards have been played so the game becomes a perfect information game. Due to a high branching factor and uncertainty of the first phase, it is challenging to describe a strategy that would lead to a victory. Such a strategy could combine different tactical approaches which humans normally use when they play a game.

3.2 Monte Carlo Tree Search

The MCTS algorithm aims to find the best possible decision in a large search space using an asymmetric tree-building process in which the most promising part of the tree is selected and iteratively exploited. The number of iterations can be limited by time or by available computational power [3]. The whole process of building a tree can be divided into four steps:

1. Selection - Starting from the root node, an unvisited child node is selected according to some selection policy. Our selection policy is UCT (Upper Confidence Bounds);

2. Expansion - A new child node is added to the tree;

3. Simulation - A move is chosen randomly and the result of this move is calculated;

4. Back-propagation - The result of the simulation is returned to the parent nodes and their values are updated. [3, 5]

3.3 Multi-armed Bandit Problems and Upper Confidence Bounds on Trees (UCT)

Multi-armed bandit problems are a subclass of decision making problems in which a choice must be made between a number of actions (i.e. arms of the bandit) in order to maximize rewards cumulatively by incrementally taking the optimal action [1]. A stochastic policy based on past rewards is used to determine which bandit should be played [2]. The policy must minimize regret - the estimated loss resulting from choosing a sub-optimal bandit. [1]

A balance needs to be achieved between the exploitation of a promising bandit and exploration of other potential bandits which may ultimately provide greater rewards, a problem termed the exploitation-exploration dilemma [1]. A common solution is to apply an upper confidence bound (UCB) to previously observed rewards to ensure that the selection process fairly explores all arms. A simple version of this, UCB1, is as follows:

$$UCB1 = \overline{X_j} + \sqrt{\frac{2ln\,n}{n_j}}$$

Where $\overline{X_j}$ is the average reward for this arm (total cumulative reward divided by the number of times this arm was visited), n is the total number of visits, and n_j is the number of times arm j has been visited [1]. UCB1 models regret as function which grows at a logarithmic rate uniformly over n [1]. The first term ensures the exploitation of high reward arms while the second term represents the upper bound on regret for this arm [1]. Its inclusion ensures less visited arms will eventually be explored as well because as visits to the arm increase, increasing the denominator, the influence of the term decreases[1]. Alternately, when other arms are visited, this term increases due to the increase of the numerator, which in turn increases the likelihood that it will be chosen in a future iteration [1, 2].

The Upper Confidence Bounds on Trees (UCT) is a further development of the UCB policy specifically for trees described by Kocsis and Szepesv [4]. Rewards drift in time for nodes below the current node due to the changing sampling probabilities[2]. This drift must be accounted for and so for the UCT algorithm, the exploration term is replaced resulting in the following :

$$UCT = \overline{X_j} + 2C\sqrt{\frac{ln\,n}{n_j}}$$

Where C is a constant term, sometimes referred to as a "tuning constant" [2]. For an adequately large C, the UCT algorithm provides consistent payoff estimates despite the drift. Further, the constant can be adjusted for use with different enhancements to the UCT algorithm. [2]

4 Research Question

Monte Carlo Tree Search is predominately used in the game of Go. The famous robot, Alpha-Go, which applies the MCTS algorithm, beats talented Go players around the world. Hence, the MCTS algorithm is now considered to be the best algorithms for Go games. We would like to investigate whether it performs optimally when applied in the game of Schnapsen.

The main goal of our research is testing the hypothesis that the MCTS algorithm applied in the Schnapsen game performs considerably better in comparison with other approaches. In particular, our research aims to analyze the rate of wins of MCTS and to prove the claim that the MCTS algorithm can win a series of games in seventy percent of the cases or, in other words, the majority of the time. We are also interested in whether we can improve MCTS by changing some measurement or policy it uses. We find two main directions to test - the number of iterations and the selection policy.

We have developed this hypothesis after studying the MCTS algorithm and its effects. Two main opponents of the MCTS bot are mybot and the rdeep bot. We think that the MCTS bot should win most of the time against mybot because mybot is a rule-based bot. It implements different strategies and applies one of them depending on the phase of the game, whose turn it is, and the current state of the hand. Rule-based means that the bot only follows the rules in order without taking other possible situations into consideration. In addition, during the first phase of the game, mybot only evaluates the present state in order to make a choice and it looks just one step ahead when it is in the second phase. Machine-learning bots based on rdeep, rand, bully, and mybot also consider moves only one step ahead, whereas the MCTS bot looks ahead until the termination of the game and thus has more information when choosing a best move.

As for the rdeep bot, the MCTS algorithm should perform better because the rdeep bot employs a Monte Carlo sampling method which allows it to look six steps ahead and evaluate the state of the game with a heuristic function. The heuristic function calculates the ratio of the player's points to the total amount of points of the player and the opponent. The state is then compared with other sample states and the move with the best score is returned. Comparatively, the MCTS algorithm actually simulates the completed game. After the simulation is completed and the winner is known, the winning score obtained at the end of the game is assigned to the root-state nodes on this path. We think that these actions give MCTS a much more accurate estimation on the possibility of winning with a specific move.

For improving MCTS, we think that the Upper Confidence Bound policy will outperform the random policy because it provides a systematic way to evaluate moves. In addition, performing more iterations will increase the accuracy of the estimation of the probability of winning with a specific move.

5 Experimental Setup

In order to test the hypothesis, we first investigated which is the strongest bot among MCTS's opponents. This was achieved by playing rdeep, bully, rand, and mybot against each other in a round-robin competition. We then played the MCTS bot against the other four bots and against the machine learning versions of these bots.

For conducting the research we used several bots:

- MCTS bot[3] - This bot implements MCTS algorithm.
  First, the bot uses a tree as the data structure. The tree node contains its parent and children nodes, the move that the player chooses, the id of this player, the number of wins with this move, and the number of times that this node was visited during the iterations of the simulation. In addition, the methods inside the Node class include: get untried moves, UCB1 select, add, and update. In the get untried moves method, the bot gets all the legal moves that have not been tried (without a win-or-lose result). In the select child method, the bot applies the UCB1 formula to calculate which is the best child to choose. In the update method, the bot checks the result by calling the function state.winning(), if the player wins with this move, we update the node's and the parents node's win-score with +1.
  In the Monte Carlo Tree Search, there are exactly 4 parts, which are Select, Expend, Simulate, and Back-propagate. If the game is not finished, the bot will do selection, expansion, and simulation in order. For Select, the bot will choose its 'best' child when the node is fully expanded. For Expand, the bot checks for the untried moves and adds a new child with that untried move to the tree. For Simulate, the bot simulates the game from the current state until it is finished. Lastly, when the game is finished, the bot calls the update method inside the Node to update the score from the leaf to root.

  The bot checks which phase it is in. For phase 1, the bot makes an assumption about the following state and for phase 2, the bot already has perfect information about the state. After that, the bot receives the best move from the Monte Carlo Tree Search method. It runs a certain number of iterations and returns the move with the largest number of visits (which means the possibility of winning with this move is the highest among all the moves).

- mybot - This is a rule-based bot which combines different strategies and applies the most suitable one in order to enhance its performance. The bot distinguishes between two phases of the game and whether a player is a leader or not. In the first phase of the game the bot is capable of exchanging a trump card with a Jack and declaring a marriage if a player leads the trick. In addition, the best move can be chosen according to the knowledge based reasoning which employs in priority order the strategy

to play Ace or the strategy to play Jack (excluding trump cards). If the player has to reply to the move of the opponent, the bot evaluates all cards it has and chooses the card of the same suit with the maximum rank which can beat the opponent's card. If such a card does not exist, two knowledge based strategies are applied consecutively: Play Jack and Play Cheap (Jack, Queen or King), both of which exclude trump cards from a choice of possible moves. In both cases (player is on lead or not), if none of the strategies could be applied then a random move is returned. In the second phase of the game, the minimax algorithm with alpha-beta pruning is used. To sum up, this bot tries to incorporate some winning strategies for Schnapsen [9] which are applicable to humans and could be used by real Schnapsen players.

- rand bot - This bot was provided by the instructor in the framework. The bot performs a move randomly.
- bully bot - This bot was provided by the instructor in the framework. This bot plays the trump card if it has one. Otherwise, it plays with the card of the same suit as the opponent's card. If the bully bot does not have trumps and can't follow the suit, it calculates the card with the highest rank in the hand and plays that card.
- rdeep bot - This bot was provided by the instructor in the framework. This bot uses the Monte Carlo strategy by sampling N random games from a given move, ranking all moves, and then choosing the move with the highest value. Ranking of moves is performed by averaging the heuristic values of the resulting states. This bot wins most of the games if it plays against rand or bully.

  We test the performance of the MCTS algorithm by running a series of games between the MCTS bot and four other bots. In order to compare the effect of MCTS with other bots more thoroughly, we have also implemented four machine learning bots based on mybot, rand bot, bully bot, and rdeep bot.

6 Results

In the experiment part, we run the tournament.py with all opponent bots and 500 games for each pair. We take the bot MCTS [3] (1000 iteration and UCB1 formula to choose a child) as the main bot to play against different default bots (with machine learning and without) and the bot we made using a knowledge based method and the alpha-beta strategy (mybot).

First we run the rdeep, bully, rand and mybot with 500 games against each other. From the result, we can confirm that rdeep is the strongest bot and that mybot follows after rdeep in strength ranking.

Result	rand	mybot	bully	rdeep
Win	477	857	502	1164

6.1 Test Against Default Bots

This test helps to determine if MCTS performs better than other bots.
From the chart below we can observe clearly that the bot using Monte Carlo Tree Search outperforms all other bots.

Mybot, which uses improved knowledge base and alpha-beta pruning (more efficient than a pure min-max strategy), performs better than the rand and bully bot. However, it is not competitive with MCTS.

Rdeep is a bot who takes samples N random games following from a given move and ranks the moves by averaging the heuristics of the resulting states. It is defeated in two-thirds of the experiments.

Result/Against	rand	mybot	bully	rdeep
Win	435	395	460	334
Lose	65	105	40	166
Win rate(500 games)	0.87	0.79	0.92	0.668

6.2 Test Against Machine-learning Bot

This test is used to check whether machine-learning improves the performance of the bot and whether MCTS still performs better.

According to this chart below, we can easily discover that Machine Learning improves the performance of rand, bully, and mybot. However, it does not improve the rdeep bot, which is interesting. We think this is because the machine-learning bot looks ahead for only one step while the rdeep bot looks ahead for six steps in one sample state. In addtion, MCTS still beats all the machine-learning bots based on rand, bully, mybot, and rdeep.

Result/Against	ml-rand	ml-mybot	ml-bully	ml-rdeep
Win	374	361	409	358
Lose	126	139	91	142
Win rate(500 games)	0.748	0.722	0.818	0.716

The chart below shows the combined result of the performance of the MCTS algorithm against other strategies. It demonstrates clearly that the win rate of the bot implementing the MCTS algorithm is much higher than any of its opponents.

6.3 Test with Different Number of Iterations

A Monte Carlo Tree Search performs iterations starting from the root state. It then selects one of the nodes to expand and simulate. Finally, it back-propagates from the expanded node and works back to the root-state node. This test is conducted to check whether the number of iterations can affect the performance.

From the result of the experiment it can be seen that the bot with more iterations wins more than the one with fewer iterations. However, more iterations requires the bot to take more time to generate a decision on which card to play. This is one drawback of the increased amount of iterations.

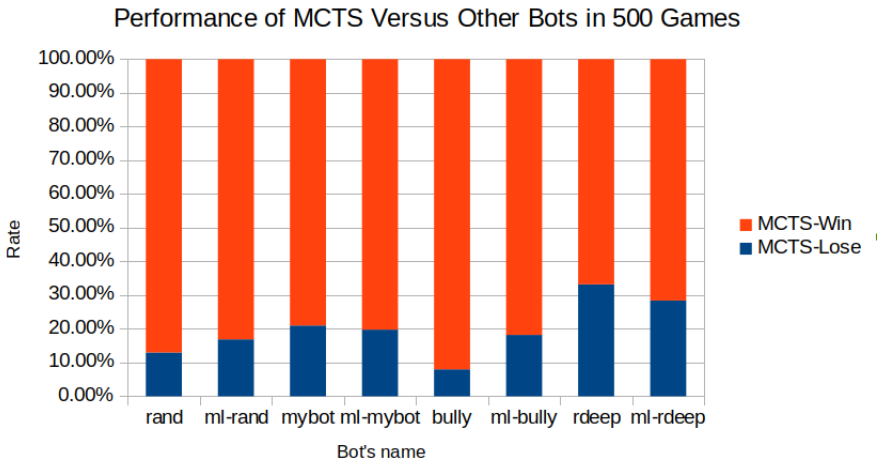

Figure 1: chart of winning rate for MCTS

Result/Against	MCTS-10 iteration	MCTS-100 iteration
Win	353	266
Lose	147	234
Win rate(500 games)	0.704	0.532

6.4 Test with Random Selection Policy

For this test, we check whether the selection policy has influence on the performance of the MCTS bot. Upper Confidence Bound is a common solution to ensure that the selection process considers each node equally, without overestimation or underestimation. We compare one bot which uses Upper Confidence Bounds as the select policy against another bot that randomly selects a child.

The number of games won by the bot with the UCB1 formula is 419 out of 500 games, which is significantly higher than that of the bot with a random policy (only 81 out of 500 games). This result shows that Upper Confidence Bound is a strong selection policy which fairly selects the 'best' child.

Result/Against	MCTS-(randomly choosing a child)
Win	419
Lose	81
Win rate(500 games)	0.838

7 Findings

The MCTS bot wins in a majority of cases when it plays against four other default bots. In particular, it wins in seventy-nine percent of games against mybot, in eighty-seven percent of games against rand and in ninety-two percent

of games against bully. This result confirms our hypothesis that the MCTS algorithm can defeat other bots in at least seventy percent of games. Compared to the rdeep bot, the MCTS bot performed a bit lower than seventy percent as it won in sixty seven percent of the games. However, this result is still very close to our hypothesis claim and we have also shown that rdeep is the strongest bot among all other bots. That is why this result still demonstrates that MCTS algorithm outperforms the Monte Carlo method implemented in rdeep.

Applying the machine learning technique to rand, bully, and mybot did improve their performance but the MCTS bot was still able to win against them in the majority of cases. For example, the percentage of wins against bully decreased from ninety-two percent to eighty-two percent. Perhaps the most interesting result was obtained while testing the MCTS bot against the machine learning rdeep bot. The MCTS bot won in seventy-two percent of the games and thus confirmed our hypothesis about the performance of the MCTS algorithm. The degraded behaviour of the rdeep machine learning bot compared to the pure rdeep bot can be explained by the fact that the machine learning rdeep bot looks only one step ahead whereas the rdeep bot looks six steps ahead. Thus looking more steps ahead could be more beneficial than training a bot with machine learning.

We have also found that the MCTS bot performs much better if the number of iterations increases. With one thousand iterations, the MCTS bot outperforms all other bots but the price to pay for this increase in wins is a considerable slowdown of the execution process.

From the obtained results, it can be also concluded that Upper Confidence Bound selection policy shows much better performance and thus should be preferred to the random policy. We have not implemented other selection policies except these two but researching the effect of other selection policies on performance of MCTS algorithm could be an attractive subject for further studies.

8 Conclusion

This paper has described the Monte Carlo Tree Search algorithm and one of its most wide-spread methods of selection policy - Upper Confidence Bound. The application of this algorithm allowed a computer to compete on a high level with professional human players in the ancient Chinese game of Go, which was previously viewed as an immense challenge for Artificial Intelligence. In order to test the performance of the MCTS algorithm, we have implemented a bot called MCTS which combined all its key components and played it against a number of other different bots. In particular, two of the main opponents of the MCTS bot were a rule-based bot called mybot and a bot employing the Monte Carlo algorithm called rdeep. Mybot used a combination of different strategies which are also extensively used by human players of the game Schnapsen. The rdeep bot applied a Monte Carlo technique which sampled a predefined amount of random games from a given move and then chose the best move according to

their ranks.

The main goal of this research was to test the claim that the MCTS bot, which applied MCTS algorithm, can win the majority of games when played against other bots, namely, in at least seventy percent of the cases. We have conducted a series of experiments in order to confirm this claim. For instance, the MCTS bot was run in a tournament against all other bots and against machine-learning versions of these bots. The obtained results have shown that the MCTS algorithm outperforms other strategies by winning in the majority of games, specifically from seventy-nine to ninety-two percent of cases. As for the Monte Carlo rdeep bot, the MCTS bot demonstrated a slightly lower winning statistics by defeating it in sixty-seven percent of the games. However, this result was improved to seventy-one percent when the MCTS bot was tested against a machine-learning version of rdeep bot. This degraded performance of the machine learning version of rdeep bot could be explained by the decreased number of steps which this bot looks ahead in its machine learning version. Apparently, looking ahead for the next several states provides the bot with a more perfect knowledge of the state of the game. In the tournament against other machine learning bots, the winning rate of the MCTS bot has decreased, ranging from seventy-two to eighty-two percent of the games. However, the overall performance of the MCTS bot was still above the claimed level of seventy percent thus proving the efficiency of applying the MCTS algorithm in the game of Schnapsen.

The results of the experiment have also favoured the usage of the Upper Confidence Bound selection policy in the MCTS algorithm to the random selection policy. In particularly, two MCTS bots, each implementing one of the selection policies, were played against each other. The bot with the Upper Confidence Bound selection policy defeated its opponent in eighty-three percent of the games. In addition, results have also shown that the winning rate of the MCTS algorithm increases considerably with the growth of the number of iterations where one iteration represents one complete simulation of the game process. It can be concluded that the MCTS algorithm has demonstrated a significant win rate in the game of Schnapsen against different types of opponents that confirms the efficiency of this method in the game environment.

9 Bibliography

[1] Browne, C., E.J. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo Tree Search methods, IEEE Trans. Comput. Intell. AI Games 4 (1), 2012, pp. 1–43.

[2] Gelly, S., L. Kocsis, M. Schoenauer, D. Silver, C. Szepesvari, O. The grand challenge of computer Go: Monte Carlo Tree Search and Extensions. Communications of the ACM, 55 (3), 2012, pp. 106-113

[3] Lewis, B. An ISMCTS AI for the card game Schnapsen. GitHub, 2019. https://github.com/tetraptych/synapsen

[4] Kocsis, L., C. Szepesvári. Bandit based Monte-Carlo planning, in: J. Fürnkranz, T. Scheffer, M. Spiliopoulou (Eds.), Euro. Conf. Mach. Learn., Springer, Germany, 2006, pp. 282–293.

[5] Powley, E.J., P. I. Cowling, D. Whitehouse. Information capture and reuse strategies in Monte Carlo Tree Search, with applications to games of hidden information. Artificial Intelligence 217, 2014, pp. 92–116.

[6] Special issue on Monte Carlo techniques and computer Go. C.S. Lee and M. Muller, and O. Teytaud, (eds). IEEE Trans. Comput. Intell. AI in Games, 2 (2010).

[7] Schlobach, S. Intelligent-systems-course Schnapsen-game. GitHub, 2019. http://github.com/intelligent-systems-course/schnapsen

[8] Schnapsen and Sixty-Six rules. Psellos, March 2, 2013 .
http://psellos.com/schnapsen/rules.html

[9] Tompa, M. Winning strategy for Schnapsen. Psellos, 2013 - 2014.
http://psellos.com/schnapsen/strategy.html

[10] Vodopivec, T., S. Samothrakis, B. Šter. On Monte Carlo Tree Search and Reinforcement Learning. Journal of Artificial Intelligence Research, 60 (1), 2017, pp. 881-936.

10 Appendix

10.1 Worksheet 1

10.1.1 Question 1

Rdeep

```
Results:
    bot <bots.rand.rand.Bot instance at 0x7f81e8737b48>: 5 wins
    bot <bots.bully.bully.Bot instance at 0x7f81e8737ab8>: 7 wins
    bot <bots.rdeep.rdeep.Bot instance at 0x7f81e873f9e0>: 18 wins
```

10.1.2 Question 2

First Bully checks for trumps in the hand, creates an array of trumps, if array is not empty plays the first one in this array.

Second, if Bully doesn't have any trumps, it will create an array of the same suit as the card played by the other player. If the array is not empty, it will play the first one in the array.

Last, if Bully has no trumps and can't follow suit, it calculates the card with the highest rank in the hand and plays that card.

10.1.3 Question 3

The hill-climbing strategy looks one move ahead using a greedy algorithm. This chooses a move that maximizes scores during the next round. However, this may not be an optimal move for the game as a whole. For example, the hill-climbing strategy may play with high ranked trumps in beginning of the game in order to collect more points, ending up with no trumps in the second phase of the game so that the opponent can easily win by holding more trumps.

10.1.4 Question 4

In a tournament of 30 games between rdeep and rand, rdeep wins 28 games which is 93% of the trails. This number is statistically significant to prove that rdeep performs better than rand.

```
Results:
    bot <bots.rand.rand.Bot instance at 0x7f26a8977cb0>: 2 wins
    bot <bots.rdeep.rdeep.Bot instance at 0x7f26a8977b48>: 28 wins
```

10.1.5 Question 5

```
def get_move(self, state):
        # type: (State) -> tuple[int, int]
        # All legal moves
        moves = state.moves()
        chosen_move = moves[0]

        for index, move in enumerate(moves):
        if move[0] is not None and move[0] % 5 <= chosen_move[0] % 5:
                chosen_move = move

        # Return a random choice
        return chosen_move
Results:
    bot <bots.rand.rand.Bot instance at 0x7f2354737cb0>: 12 wins
    bot <bots.mybot.mybot.Bot instance at 0x7f23547375a8>: 18 wins
```

10.1.6 Question 6

This heatmap shows the number of wins corresponding to each player in respect to the lowmove probability. In this case the lowmove probability corresponds to moves which are non-trump suits. Blue color means that more games were won by player 2, red color means that more games were won by player 1. The resulting heatmap displays a random pattern and and suggests that no correlation exists between the lowmove probability and number of wins.

10.1.7 Question 7

Code:
value, m = self.value(next_state, depth+1)
Results:
bot <bots.rand.rand.Bot instance at 0x7f4ba92eac68>: 9 wins
bot <bots.minimax.minimax.Bot instance at 0x7f4ba92f13b0>: 21 wins

10.1.8 Question 8

Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Agreed.
Done. time Minimax: 0.00448242823283, time Alphabeta: 0.00382431348165.
Alphabeta speedup: 1.17208703061

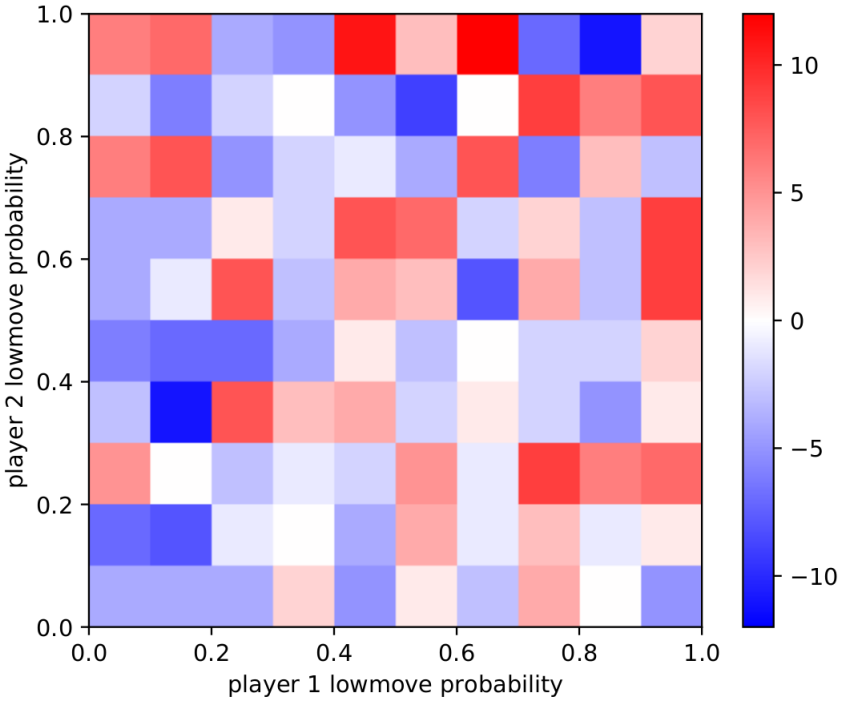

Figure 2: heatmap

10.1.9 Question 9

These implementations use heuristic based on the function ratio_points of general utility functions. The function ratio_points returns the ratio of the player's points to the sum of the player's and opponent's points. After two is multiplied by this value and one is subtracted from it. Because of these calculations heuristic can return an evaluation for the given state between -1 and 1. For example, if both players have the same amount of points, then ratio_points returns 0.5. 0.5 *2 - 1 = 0. It means that state is evaluated as 0 so that positions of the players are equal.

Our improved heuristic is based on the existing ones but also takes into account the number of trumps the player has. If the number of trumps the player has is higher then the number of trumps in opponent's hands, the player receives 0.5 bonus points. If this number is smaller, 0.5 will be subtracted from estimated value.

```
def heuristic(state):
        trump = state.get_trump_suit()
        player_1 = state.hand(1)
```

```
        player_2 = state.hand(2)

player_1_trumps = []
for card in player_1:
    if util.get_suit(card) == trump:
                    player_1_trumps.append(card)

player_2_trumps = []
for card in player_2:
     if util.get_suit(card) == trump:
                     player_2_trumps.append(card)

if len(player_1_trumps) > len(player_2_trumps):
      return util.ratio_points(state, 1) * 2.0 - 1.0 + 0.5 , None

if len(player_1_trumps) < len(player_2_trumps):
      return util.ratio_points(state, 1) * 2.0 - 1.0 - 0.5, None

if len(player_1_trumps) == len(player_2_trumps):
     return util.ratio_points(state, 1) * 2.0 - 1.0 , None
```

Results of tournament between alphabet and mybot:

```
Results:
    bot <bots.mybot.mybot.Bot instance at 0x7f93c2d06e60>: 20 wins
    bot <bots.alphabeta.alphabeta.Bot instance at 0x7f93c2d06f38>: 10 wins
```

10.2 Worksheet 2

10.2.1 Question 1

```
kb.add_clause(~B, ~C)
```

10.2.2 Question 2

```
kb.add_clause(A,B)
kb.add_clause(~B,A)
kb.add_clause(~A,C)
kb.add_clause(~A,D)
```

Result possible models:
A: True, C: True, B: False, D: True
A: True, C: True, B: True, D: True
So knowledge base can contain only 2 models which differ by B (True or False),
and A, C, D must be True.

10.2.3 Question 3

If it's possible to find at least one integer value when this model is true it means that this model is satisfiable. Otherwise it's unsatisfiable.
When x = y = 2, it holds for all these three constraints.

10.2.4 Question 4

```
[x = y] = True, [x + y > 2] = True, [x + y < 5] = True
[x = y] = True, [x + y > 2] = True, [x + y >= 5] = True
[x = y] = True, [x + y <= 2] = True, [x + y < 5] = True
```

10.2.5 Question 5

```
2 models were returned:
{[y + x < 5]: True, [y + x > -5]: True, [y + x < -2]: True, [y + x > 2]:
False, [(-y) + x == 0]: True}
{[y + x > -5]: True, [y + x < 5]: True, [y + x < -2]: False, [y + x > 2]:
True, [(-y) + x == 0]: True}
```

10.2.6 Question 6

The strategy of playing ace first is defined by creating a variable A and initializing it to A0, A5, A10, A15. Play Ace is the strategy to play ace first: for all x PA(x) <-> A(x). In order to check the correctness of strategy we added a clause kb.add_clause(-PA0), so that when we run the test it returns false meaning that a move is entailed by your knowledge base.

10.2.7 Question 7

To implement the strategy of a cheap card we defined a variable C and initialized it to C4, C9, C14, C19 (for jack), C3, C8, C13, C18 (for queen), C2, C7, C12, C17 (for king). We added clauses to knowledge base according to the rule PJ(x) <-> J(x) or Q(x) or K(x). Cheap card strategy plays a card when it is a jack, a queen or a king. In order to check the correctness of strategy we added a clause kb.add_clause(-PC4), so that when we run the test it returns false meaning that a move is entailed by your knowledge base.

10.2.8 Question 8

```
player1: <bots.kbbot.kbbot.Bot instance at 0x7f7f80a18f38>
player2: <bots.rand.rand.Bot instance at 0x7f7f80a1f320>

Start state: The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 0, pending: 0
The trump suit is: D
Player 1's hand: KC QC 10D 10H JH
```

```
Player 2's hand: AC 10C JC QD AH
There are 10 cards in the stock
*   Player 1 plays: KC
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 0, pending: 0
The trump suit is: D
Player 1's hand: KC QC 10D 10H JH
Player 2's hand: AC 10C JC QD AH
There are 10 cards in the stock
Player 1 has played card: K of C

*   Player 2 plays: 10C
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 14, pending: 0
The trump suit is: D
Player 1's hand: QC 10D 10H JH AS
Player 2's hand: AC JC QD AH KS
There are 8 cards in the stock

*   Player 2 plays: KS
The game is in phase: 1
Player 1's points: 0, pending: 0
Player 2's points: 14, pending: 0
The trump suit is: D
Player 1's hand: QC 10D 10H JH AS
Player 2's hand: AC JC QD AH KS
There are 8 cards in the stock
Player 2 has played card: K of S

Strategy Applied
*   Player 1 plays: AS
The game is in phase: 1
Player 1's points: 15, pending: 0
Player 2's points: 14, pending: 0
The trump suit is: D
Player 1's hand: QC 10D 10H KH JH
Player 2's hand: AC JC QD JD AH
There are 6 cards in the stock

*   Player 1 plays: KH
The game is in phase: 1
Player 1's points: 15, pending: 0
Player 2's points: 14, pending: 0
The trump suit is: D
```

```
Player 1's hand: QC 10D 10H KH JH
Player 2's hand: AC JC QD JD AH
There are 6 cards in the stock
Player 1 has played card: K of H

*   Player 2 plays: JD
The game is in phase: 1
Player 1's points: 15, pending: 0
Player 2's points: 20, pending: 0
The trump suit is: D
Player 1's hand: QC 10D 10H JH QS
Player 2's hand: AC JC QD AH QH
There are 4 cards in the stock

*   Player 2 plays: JC
The game is in phase: 1
Player 1's points: 15, pending: 0
Player 2's points: 20, pending: 0
The trump suit is: D
Player 1's hand: QC 10D 10H JH QS
Player 2's hand: AC JC QD AH QH
There are 4 cards in the stock
Player 2 has played card: J of C

*   Player 1 plays: 10D
The game is in phase: 1
Player 1's points: 27, pending: 0
Player 2's points: 20, pending: 0
The trump suit is: D
Player 1's hand: QC 10H JH 10S QS
Player 2's hand: AC KD QD AH QH
There are 2 cards in the stock

*   Player 1 plays: QS
The game is in phase: 1
Player 1's points: 27, pending: 0
Player 2's points: 20, pending: 0
The trump suit is: D
Player 1's hand: QC 10H JH 10S QS
Player 2's hand: AC KD QD AH QH
There are 2 cards in the stock
Player 1 has played card: Q of S

*   Player 2 plays: QD
The game is in phase: 2
Player 1's points: 27, pending: 0
```

```
Player 2's points: 26, pending: 0
The trump suit is: D
Player 1's hand: QC AD 10H JH 10S
Player 2's hand: AC KD AH QH JS
There are 0 cards in the stock

*   Player 2 plays: AC
The game is in phase: 2
Player 1's points: 27, pending: 0
Player 2's points: 26, pending: 0
The trump suit is: D
Player 1's hand: QC AD 10H JH 10S
Player 2's hand: AC KD AH QH JS
There are 0 cards in the stock
Player 2 has played card: A of C

*   Player 1 plays: QC
The game is in phase: 2
Player 1's points: 27, pending: 0
Player 2's points: 40, pending: 0
The trump suit is: D
Player 1's hand: AD 10H JH 10S
Player 2's hand: KD AH QH JS
There are 0 cards in the stock

*   Player 2 plays: QH
The game is in phase: 2
Player 1's points: 27, pending: 0
Player 2's points: 40, pending: 0
The trump suit is: D
Player 1's hand: AD 10H JH 10S
Player 2's hand: KD AH QH JS
There are 0 cards in the stock
Player 2 has played card: Q of H

*   Player 1 plays: 10H
The game is in phase: 2
Player 1's points: 40, pending: 0
Player 2's points: 40, pending: 0
The trump suit is: D
Player 1's hand: AD JH 10S
Player 2's hand: KD AH JS
There are 0 cards in the stock

Strategy Applied
*   Player 1 plays: AD
```

```
The game is in phase: 2
Player 1's points: 40, pending: 0
Player 2's points: 40, pending: 0
The trump suit is: D
Player 1's hand: AD JH 10S
Player 2's hand: KD AH JS
There are 0 cards in the stock
Player 1 has played card: A of D

*   Player 2 plays: KD
The game is in phase: 2
Player 1's points: 55, pending: 0
Player 2's points: 40, pending: 0
The trump suit is: D
Player 1's hand: JH 10S
Player 2's hand: AH JS
There are 0 cards in the stock

*   Player 1 plays: 10S
The game is in phase: 2
Player 1's points: 55, pending: 0
Player 2's points: 40, pending: 0
The trump suit is: D
Player 1's hand: JH 10S
Player 2's hand: AH JS
There are 0 cards in the stock
Player 1 has played card: 10 of S

*   Player 2 plays: JS
The game is in phase: 2
Player 1's points: 67, pending: 0
Player 2's points: 40, pending: 0
The trump suit is: D
Player 1's hand: JH
Player 2's hand: AH
There are 0 cards in the stock
Game finished. Player 1 has won, receiving 1 points.
```

10.2.9 Question 9

```
Against a simple knowledge bot playing jack always:
Results:
        bot <bots.kbbot.kbbot.Bot instance at 0x7f0002401878>: 17 wins
        bot <bots.kbjack.kbjack.Bot instance at 0x7f00024072d8>: 13 wins

Against a simple knowledge bot playing ace always:
```

```
Results:
        bot <bots.kbbot.kbbot.Bot instance at 0x7f8d71b91878>: 16 wins
        bot <bots.kbace.kbace.Bot instance at 0x7f8d71b972d8>: 14 wins

Against rand:
Results:
        bot <bots.kbbot.kbbot.Bot instance at 0x7f1fbfd66878>: 16 wins
        bot <bots.rand.rand.Bot instance at 0x7f1fbfd6c2d8>: 14 wins
```

10.3 Worksheet 3

10.3.1 Question 1

```
In function value:
value = self.heuristic(next_state)

In function features:
def features(state):
        # type: (State) -> tuple[float, ...]
        """
        Extract features from this state. Remember that every feature vector retu

        :param state: A state to be converted to a feature vector
        :return: A tuple of floats: a feature vector representing this state.
        """

        feature_set = []

        perspective = state.get_perspective()

        # Convert the card state array containing strings, to an array of intege
        # The integers here just represent card state IDs. In a way they can be
        # thought of as arbitrary, as long as they are different from each other
        perspective = [card if card != 'U' else (-1) for card in perspective]
        perspective = [card if card != 'S' else 0 for card in perspective]
        perspective = [card if card != 'P1H' else 1 for card in perspective]
        perspective = [card if card != 'P2H' else 2 for card in perspective]
        perspective = [card if card != 'P1W' else 3 for card in perspective]
        perspective = [card if card != 'P2W' else 4 for card in perspective]

        feature_set += perspective

        # Add player 1's points to feature set
        p1_points = state.get_points(1)
        feature_set.append(p1_points)

        # Add player 2's points to feature set
```

```
        p2_points = state.get_points(2)
        feature_set.append(p2_points)

        # Add player 1's pending points to feature set
        p1_pending_points = state.get_pending_points(1)
        feature_set.append(p1_pending_points)

        # Add plauer 2's pending points to feature set
        p2_pending_points = state.get_pending_points(2)
        feature_set.append(p2_pending_points)

        # Get trump suit
        trump_suit = state.get_trump_suit()

        # Convert trump suit to id and add to feature set
        # You don't need to add anything to this part
        suits = ["C", "D", "H", "S"]
        trump_suit_id = suits.index(trump_suit)
        feature_set.append(trump_suit_id)

        # Add phase to feature set
        phase = state.get_phase()
        feature_set.append(phase)

# Add stock size to feature set
        stock_size = state.get_stock_size()
        feature_set.append(stock_size)

        # Add leader to feature set
        leader = state.leader()
        feature_set.append(leader)

        # Add whose turn it is to feature set
        whose_turn = state.whose_turn()
        feature_set.append(whose_turn)

        # Add opponent's played card to feature set
        opponents_played_card = state.get_opponents_played_card()

        # You don't need to add anything to this part
        opponents_played_card = opponents_played_card if opponents_played_card i
        feature_set.append(opponents_played_card)

        # Return feature set
        return feature_set
```

```
Results of tournament:
    bot <bots.ml.ml.Bot instance at 0x7fab8145d320>: 41 wins
        bot <bots.bully.bully.Bot instance at 0x7fab6ff92c20>: 22 wins
        bot <bots.rand.rand.Bot instance at 0x7fab6ff92cb0>: 27 wins
```

10.3.2 Question 2

Rdeep bot was used for training, number of games was reduced for 1000

```
Results:
        bot <bots.ml.ml.Bot instance at 0x7f75cde77320>: 43 wins
        bot <bots.bully.bully.Bot instance at 0x7f75bc9acc20>: 24 wins
        bot <bots.rand.rand.Bot instance at 0x7f75bc9accb0>: 23 wins
```

10.3.3 Question 3

We made 3 models of rdeep, rand and kbbot and played them against each other.

```
Results:
        bot <bots.mlrdeep.mlrdeep.Bot instance at 0x7f9887e34320>: 35 wins
        bot <bots.mlkb.mlkb.Bot instance at 0x7f986e9764d0>: 27 wins
        bot <bots.mlrand.mlrand.Bot instance at 0x7f986e976638>: 28 wins
```

10.3.4 Question 4

2 simple features were added by us. First, we create a new variable point_difference which is the difference between points of player 1 and player 2. The higher this value the better it is for player 1. Second, we created another variable trump_points and analyzed the current cards of player 1 and count how many points with trumps he has.

```
Results of tournament against rand and bully:
        bot <bots.mlfeatures.mlfeatures.Bot instance at 0x7fd7213e2638>: 46 wins
        bot <bots.bully.bully.Bot instance at 0x7fd7213c0f80>: 21 wins
        bot <bots.rand.rand.Bot instance at 0x7fd70ff1dc68>: 23 wins
```

10.4 Code for MCTS Bot

This code is based on Lewis, B. An ISMCTS AI for the card game Schnapsen, which is an open-source online GitHub repository[3]. The code is adapted to the functions in the skeleton.

```
from api import State, util
from math import *
import random, sys

class Node:
```

```
    def __init__(self, move = None, parent = None, playerJustMoved = None):
        self.move = move
        self.parentNode = parent
        self.childNodes = []
        self.wins = 0
        self.visits = 0
        self.avails = 1
        self.playerJustMoved = playerJustMoved

        def GetUntriedMoves(self, legalMoves):
                triedMoves = [child.move for child in self.childNodes]
                return [move for move in legalMoves if move not in triedMoves]

    def UCBSelectChild(self, legalMoves, exploration = 0.7):
        legalChildren = [child for child in self.childNodes
        if child.move in legalMoves]
        s = max(legalChildren, key = lambda c: float(c.wins)/float(c.visits)
        + exploration * sqrt(log(c.avails)/float(c.visits)))
        for child in legalChildren:
            child.avails += 1
        return s

    def AddChild(self, m, p):
        n = Node(move = m, parent = self, playerJustMoved = p)
        self.childNodes.append(n)
        return n

    def Update(self, terminalState):
        self.visits += 1
        if self.playerJustMoved is not None:
            player, points = terminalState.winner()
            if(player == self.playerJustMoved):
                self.wins += 1

class Bot:
    def __init__(self):
        pass

    def get_move(self, state):
        if state.get_phase() == 1:
            sample_state = state.make_assumption()
        else:
            sample_state = state
        best_move = ISMCTS(sample_state, 1000)
        return best_move # Return the best scoring move
```

```
def ISMCTS(state, itermax):
    rootnode = Node()
    for i in range(itermax):
        node = rootnode
        state_copy = state.clone()

        # Select
        while state_copy.finished() is False
        and node.GetUntriedMoves(state_copy.moves()) == []:
            node = node.UCBSelectChild(state_copy.moves())
            state_copy = state_copy.next(node.move)

        # Expand
        untriedMoves = node.GetUntriedMoves(state_copy.moves())
        if untriedMoves != [] and state_copy.finished() is False:
            m = random.choice(untriedMoves)
            player = state_copy.whose_turn()
            state_copy = state_copy.next(m)
            node = node.AddChild(m, player) # add child and descend tree

        # Simulate
        while state_copy.finished() is False:
            state_copy = state_copy.next(random.choice(state_copy.moves()))

        # Backpropagate
        while node != None:
            node.Update(state_copy)
            node = node.parentNode

    return max(rootnode.childNodes, key = lambda c: c.visits).move
```

10.5 Code for Mybot

```
from api import State, util
from api import Deck
import random, load

from kb import KB, Boolean, Integer

class Bot:

    __max_depth = -1
    __randomize = True

    def __init__(self, randomize=True, depth=8):
```

```
        self.__randomize = randomize
        self.__max_depth = depth

    def get_move(self, state):

        if state.get_phase() is 1:
            moves = state.moves()
            random.shuffle(moves)

            if state.leader() is 1:
                for move in moves:
                    if move[0] is None:
                        return move
                    if state.get_pending_points(1)
                    and not self.kb_consistent(state, move, "pa"):
                        return move
                    if not self.kb_consistent(state, move, "pw"):
                        return move
                    if not self.kb_consistent(state, move, "pa"):
                        return move
                    if not self.kb_consistent(state, move, "pj"):
                        return move
                return random.choice(moves)

            played_card = state.get_opponents_played_card()
            min_value = 50
            min = None
            for move in moves:
                if (move[0] is not None) and played_card is not None
                and (util.get_suit(move[0]) == util.get_suit(played_card))
                and (move[0] < played_card):
                    if move[0] < min_value:
                        min = move
                        min_value = move[0]
            if min is not None:
                return min
            for move in moves:
                if not self.kb_consistent(state, move, "pj"):
                    if util.get_suit(move[0]) == state.get_trump_suit():
                        continue
                    return move
                if not self.kb_consistent(state, move, "pc"):
                    if util.get_suit(move[0]) == state.get_trump_suit():
                        continue
                    return move
            return random.choice(moves)
```

```
        if state.get_phase() is 2:
            val, move = self.value(state)
            return move

    def kb_consistent(self, state, move, strategy):
        kb = KB()

        load.general_information(kb)
        return kb.satisfiable()

    def value(self, state, alpha=float('-inf'), beta=float('inf'), depth = 0):
        if state.finished():
            winner, points = state.winner()
            return (points, None) if winner == 1 else (-points, None)
        if depth == self.__max_depth:
            return heuristic(state)
        best_value = float('-inf') if maximizing(state) else float('inf')
        best_move = None
        moves = state.moves()
        if self.__randomize:
            random.shuffle(moves)
        for move in moves:
            next_state = state.next(move)
            value, m = self.value(next_state, depth+1)
            if maximizing(state):
                if value > best_value:
                    best_value = value
                    best_move = move
                    alpha = best_value
            else:
                if value < best_value:
                    best_value = value
                    best_move = move
                    beta = best_value
            if beta < alpha:
                break
        return best_value, best_move

def maximizing(state):
    return state.whose_turn() == 1

def heuristic(state):
    return util.ratio_points(state, 1) * 2.0 - 1.0 , None
```