Project 3: Decision Trees for Expert Iteration (Reinforcement learning) (revised April 2020)

Shahzadi Mahaa Irshad
CSEE, University of Essex
Reg. ID: 1901079
E-mail: si19783@essex.ac.uk

Abstract—This work investigates the performance of an iterative-learning based supervised prediction model implemented with the Monte Carlo tree search (MCTS). We exploit the random rollout feature of MCTS to generate a dataset with potential states and optimal moves, solely based on the trained agent and random moves, without any expert human's data. Moreover, to perform empirical analysis, we train 20 individual decision-tree-based agents on the datasets generated to play against its previous versions. We increase the complexity of the evaluation games by introducing randomization in the first player and first move selection. We demonstrate that training the agent for five iterations yields the highest win-rate, and training for fourteen iterations yields the highest draw-rate.

Index Terms—MCTS, supervised learning, iterative-learning, decision-tree

1 Introduction

Developing game playing ability in machines has been a widely adopted method to stimulate computational intelligence for almost 60 years [1]. In 1959, the first artificially intelligent agent was built by Arthur L. Samuel to play the game of checkers [2]. Since then, several AI (artificial intelligence) based methods have been developed for various games, including backgammon, Othello, checkers, ludo, and go [3]. In this work, a category of machine learning, the supervised learning method, is employed to learn the game of tic-tac-toe iteratively. A decision tree is trained to play the best move learned on every iteration with some random moves for exploration. A Monte Carlo Tree Search (MCTS) is used with the machine learning agent to find the potential states from a given problem and the actions that will achieve them.

Several agents are created by using the dataset generated by playing with the most recent version of itself on each iteration. The first dataset is generated by using the Upper Confidence Bounds Applied to Trees (UCT) method with MCTS. To evaluate the performance, the winning rate and drawing rate is obtained by each agent playing with all the past and future versions of itself. The complexity in the final matches is increased by adding randomization in the first player and first move selection.

The following section covers the related work done in the areas involved in this research. Section 3 gives a detailed description of the methodologies adopted and the problems encountered in its implementation. It covers the UCT/MCTS method, data generation strategy, and augmentation of MCTS with supervised learning. Furthermore, Section 4 presents the empirical results obtained, followed by Section 5, where these results are discussed. Finally, the last section contains the conclusions derived from this research work.

2 Background

Several machine learning strategies have been employed in game-playing, including reinforcement learning, supervised learning, and unsupervised learning [4]. Moreover, game playing requires a tree search algorithm to find the possible states and actions for a given problem, among which the most commonly used methods include minimax, alpha-beta search, and MCTS.

The minimax [5] and alpha-beta [6] search are the most popular two-player search techniques. Several methods using the minimax search technique have also been created for multiplayer games; however, all of the techniques mentioned above require a heuristic evaluation function, which can be very difficult to create for complex games [7]. An alternate method is the MCTS algorithm, which has proven to be the most efficient tree search method from the earliest creations to date [2].

MCTS gained popularity when it was used in Google Deepmind's AlphaGo to play the game of Go in 2016 [8]-[10]. It employs the MCTS for look-ahead search and trains on two deep neural networks (NN) using knowledge from human experts. The first neural network is a policy network that narrows down the search to high probability

• *S. M. Irshad is an MSc. In Intelligent Systems and Robotics student at the University of Essex, Colchester, CO4 3SQ, UK. E-mail: si19783@essex.ac.uk*

moves [8]-[10]. The second neural network is a value network that predicts the winner of games played by the policy network itself [8]-[10]. A modification to the AlphaGo, the AlphaGo Zero, omits out the need for data from an expert human player by self-playing randomly [11]. The AlphaGo Zero uses a single neural network to train on the data generated through MCTS by taking random moves in each state.

MCTS has also been employed for more complex games with partially hidden information, such as the Hearthstone video game [12]. [12] uses MCTS to find the optimal action policy and augments it with a neural network. The agent improves the win-rate by playing itself for several iterations.

AI is a growing field, and new methodologies are being introduced gradually to outperform the state-of-the-art. This research project aims to employ the current state-of-the-art in game playing using machine learning for the game of tic-tac-toe. The MCTS is employed as a tree-search for guidance, and data is generated solely by self-playing. The game-playing agents are built using a supervised learning method, the decision tree which improves on each iteration by self-playing.

3 Methodology

I. *Tic-tac-toe game with MCTS*

A. Game playing

The tic-tac-toe game consists of a 3x3 square grid, which is filled by each of the two players on their turns. In order to win, a player has to mark three places in a row (horizontally, vertically, or diagonally). The game ends when all the nine blocks of the grid are filled.

MCTS is employed for a look-ahead search. It finds the potential states and actions for a given state of the grid. UCT/MCTS is used to generate the first dataset by repeatedly sampling a problem space randomly to develop a more accurate understanding of the best answers. The basis of the program is the MCTS code generated by [13]. The agent plays against itself for 10000 times. The working of a UCT/MCTS is explained in more detail in the next section.

B. Monte Carlo Tree Search

MCTS is a heuristic driven search algorithm that comprises of a tree search algorithm and machine learning principles of reinforcement learning. The tree search searches for the potential options from a state, and reinforcement learning identifies the best option. A tradeoff exists between exploration and exploitation; the former refers to validating the current best option, and the latter refers to looking for other good options.

The exploration for the best option is carried out by running separate simulations from a given state without affecting the actual state of the game being played. These simulations run in a 'forward model,' which enables forward planning. The process contains four phases discussed below.

The MCTS relies on four phases: selection, expansion, simulation, and backpropagation. The four phases are repeated iteratively, forming a tree, as shown in fig. 1.

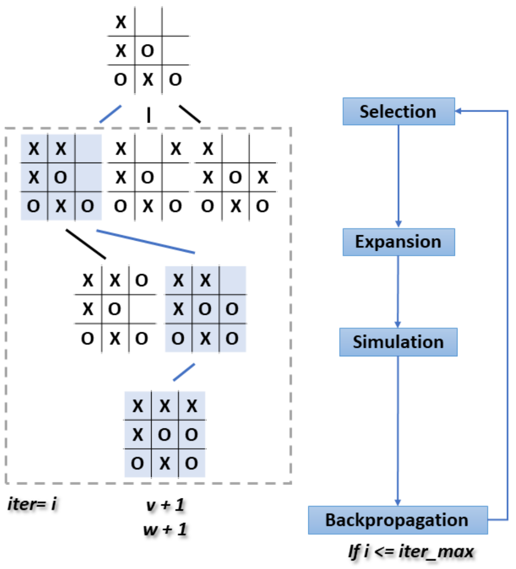


Fig. 1: The MCTS schema

Selection: In the selection phase, for a given state of the board a selection of a future state down the tree is made using the UCB1 strategy [14], [15]. The formula for UCB1 is shown in eq. 1.

$$UCB1 = \frac{w_i}{n_i} + C\sqrt{\frac{l_n N_i}{n_i}} \qquad \text{Eq.1}$$

Where, w_i is the win rate, n_i is the number of visits done on a child node i, N_i is the number of visits on the parent node from where the selection is being made, and C is a constant used to develop a balance between exploration and exploitation.

Expansion: The selected state from the first phase is exposed to reveal possible future states, one step down. The expansion is done provided that the game has not ended, and the maximum tree depth is not achieved.

Simulation: Simulation is an essential phase of the MCTS. It carries out a 'random playout' in which each node is selected randomly until the terminal node is reached.

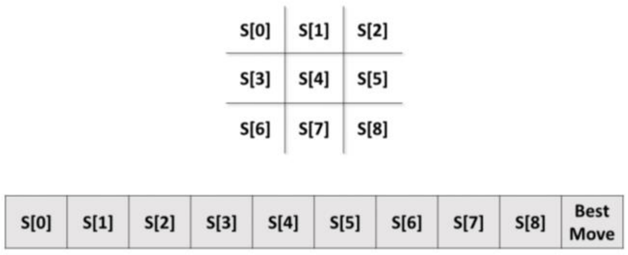
Backpropagation: At the end of each iteration, the 'visits' and 'wins' are backpropagated to the parent node. The 'visits' are simply incremented by 1 for each node that is used to reach the terminal node. The 'wins' for the given player is incremented by one if it wins and 0 in any other case.

The four steps are repeated for the maximum number of iterations provided. Once the maximum iterations are processed, the best move is selected based on the node with the highest number of visits.

To generate the first dataset, the maximum iterations for player 1 is set to 1000 and 100 for player 2. These values were varied to have the same number of iterations for both players, but it does not result in a significant improvement in the win-rate.

II. *Dataset Generation*

The dataset for this project is generated from scratch by self-playing supervised learning-based agents. To train the first supervised learning-based agent, the dataset is generated by UCT/MCTS agent playing against itself for 10000 matches. This figure is inspired by [12], where they generate data on 20000 games. Due to computational limitations in this project, this figure is reduced by 50%. For supervised learning, the number of games is further reduced to 2000 as the learning and prediction require more time.



S[0]	S[1]	S[2]
S[3]	S[4]	S[5]
S[6]	S[7]	S[8]

S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]	S[8]	Best Move

Fig. 2: Array formed for the dataset

To create homogeneity between the dataset created by MCTS and supervised learning agents, the same approach is used. An array is formed for the best action taken in each state, as shown in fig. 2. 'S[0]' to 'S[8]' stores information of the state of the game, and 'Best Move' stores the action taken by the player. The actions taken by the player under consideration are marked with '1,' and the opponent's actions are marked with '0'. This notation is reversed every time an action for player 2 is required while playing.

For each of the 'n' number of games played, all the states and actions of the player that wins are stored. As the second-best result after winning is drawing the match; therefore, data for both the players is stored when the match results in a tie.

III. *Augmenting MCTS with supervised learning*

A. Decision tree with MCTS

The state-of-the-art method used in AlphaGo uses supervised learning with MCTS to provide heuristic evaluation for state and actions. A more straightforward method for implementing supervised learning is used in this work by employing a decision tree (DT). Several DT based agents are produced through iterative learning. This program comprises of two parts: data generation and agents' competition.

The flow of the data generated can be seen in fig. 3. A total of 20 DT-based agents are trained on the dataset generated by the previous agent, augmented with MCTS for guidance. A decision tree is trained on each iteration, and it plays against itself for 2000 games to generate a dataset. The results of the games are stored in the way described in the previous section. The first dataset is used from UCT/MCTS self-play.

In the second part of the program, the dataset generated by each game set is used to train individual agents (clf1 to clf20). All of the agents developed play against each other for 200 games to evaluate performance improvement or loss.

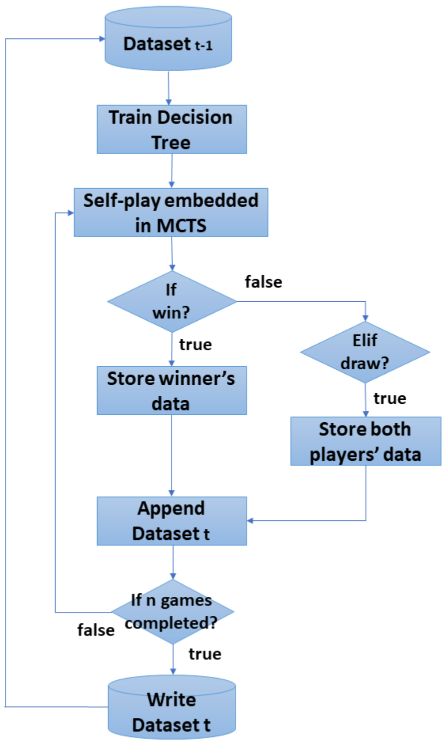


Fig. 3: Program flowchart for data generation using supervised learning.

B. Iterative learning

In UCT/MCTS, the simulations are carried by random rollouts; however, in supervised learning, these rollouts are not kept entirely random. A DT-based agent created by the previous dataset generated is used to find the best move for 90% of the iterations. The remaining 10% of the iterations use a random move from the available options. This enables to execute the learning of decision tree for maximum iterations, whereas, providing flexibility to explore more potential options.

C. Handling the imperfect DT fitting problem

As the trained decision trees are incapable of always making a perfect fit; therefore, on some occasions, it predicts a move on the board where a mark already exists. This prediction leads to an inconclusive outcome; therefore, to avoid it, a random move is suggested. The random movement is selected from the available options on the board for a given state. Although the random move does not lead to an accurate conclusion, it helps to avoid disruption during iterations.

D. Increasing game complexity

In the evaluation phase, two DT-based agents play against each other at a time. The match becomes similar and repetitive if the same player starts the game every time, resulting in the same result for each match. To genuinely evaluate the performance of the agents, the complexity of the game is increased. The first player for each of the

200 games is selected randomly, and the first move taken by that player is also chosen randomly. This allows the DT-based agents to demonstrate their performance for different types of problems.

E. Handling the randomization issue

Due to the inclusion of randomization during the competitions, the best performer can be easily mistaken for a set of 200 games. For example, if taking the first move increases the chances of winning, one of the agents can smoothly perform better than the other if it receives more chances to take the first move. Therefore, to avoid the biasness, the set of 200 games is played for 5 iterations. The mean of the winning and drawing rate for each iteration is evaluated to deduce the final outcome.

4 Results

The dataset generated by UCT/MCTS through self-playing for 10000 iterations is used to train the first DT-based agent. The DT-based agent, embedded in MCTS, self-plays for another 2000 iterations and generates a new dataset. Exactly 20 datasets are generated in this way to train 20 individual DT-based agents.

As the purpose of creating an agent based on learning from the previous agent is to improve the performance of the agent, therefore, an empirical analysis is performed to evaluate the performance loss or improvement. The 20 DT-based agents play against all past versions of itself for 1000 (200x5) games. The average winning rate and drawing rate for the games played are obtained.

Table 1 shows the winning rate obtained for each agent playing against all versions of itself. The winning percentage is displayed for the player 1 in each match. Fig. 4 illustrates the graph plotted, where the x-axis shows the second player, and the plots show the win rates of the first player. The average winning rate is shown on the y-axis.

Table 1: Winning-rates (in %) of player 1 in 20 vs. 20 agents match

Player 2

Player 1

	clf1	clf2	clf3	clf4	clf5	clf6	clf7	clf8	clf9	clf10	clf11	clf12	clf13	clf14	clf15	clf16	clf17	clf18	clf19	clf20
clf1	7.5																			
clf2	4.8	0																		
clf3	14	10	6.3																	
clf4	7.2	4.2	6.8	3.2																
clf5	5.8	5.3	7.5	0	7.1															
clf6	12	17	15	8.5	4.5	0														
clf7	5.2	17	15	4.6	7.1	3.1	5.7													
clf8	11	11	15	6.4	5.5	7.4	5.7	12												
clf9	0	16	11	0.2	13	9.1	12	6.4	0											
clf10	1.6	13	16	2.3	0	1.5	15	3.6	17	17										
clf11	0	20	6.7	3.1	12	0	3.9	1.1	13	2.9	9.4									
clf12	5.7	12	14	14	13	15	16	11	14	8.6	12	12								
clf13	5.3	12	12	0	0.5	0	18	4.1	21	10	11	16	0							
clf14	6	10	0	3.4	0	3.3	2.6	4.6	11	4.6	5.4	16	0.9	0						
clf15	10	17	5.7	2.4	0	2.8	11	5	4.9	11	12	16	0	0	7.1					
clf16	11	7.7	5.4	8	7.5	10	12	10	6.6	10	11	18	1.9	4.8	2.4	12				
clf17	7.6	5.6	12	13	5.8	10	7.2	8.3	18	18	5.8	37	3.4	13	9.2	13	5.8			
clf18	11	13	5.5	6.8	1.2	9.4	6.6	12	5.2	11	13	20	6.9	5.4	6.4	14	7.4	5.7		
clf19	2.4	6	1.7	3.9	12	5.8	1.4	3.7	8.5	17	9.2	17	14	8.3	6.6	10	4.3	5.9	0	
clf20	12	0	0	8.7	6.4	4.8	18	5.3	5.5	11	7.1	19	2	2	0	5.8	6.2	0	7.8	0

Table 2 shows the drawing rate obtained for each agent playing against all versions of itself. Fig. 5 illustrates the graph plotted, where the x-axis shows the second player, and the plots show the drawing rates of the first player. The average drawing rate is shown on the y-axis.

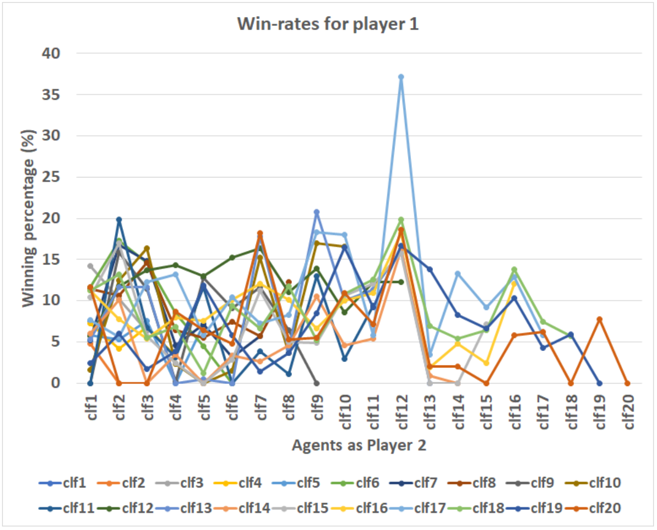


Fig. 4: Win-rates of Player 1 plotted for 20 vs. 20 agents

Table 2: Drawing-rates (in %) of 20 vs. 20 agents

Player 2

Player 1

	clf1	clf2	clf3	clf4	clf5	clf6	clf7	clf8	clf9	clf10	clf11	clf12	clf13	clf14	clf15	clf16	clf17	clf18	clf19	clf20
clf1	88																			
clf2	83	100																		
clf3	80	79	90																	
clf4	93	81	81	95																
clf5	94	89	88	100	88															
clf6	88	70	78	85	91	100														
clf7	84	76	76	86	87	66	90													
clf8	83	78	77	89	78	86	89	78												
clf9	89	54	85	88	65	74	69	71	100											
clf10	82	69	80	92	90	78	71	76	62	68										
clf11	90	80	75	84	73	82	80	89	70	77	84									
clf12	84	61	65	69	61	48	68	69	71	73	80	78								
clf13	95	86	80	100	96	98	81	94	68	86	87	65	100							
clf14	94	83	91	90	87	85	81	82	83	86	83	81	90	100						
clf15	79	78	89	84	95	79	74	89	82	72	82	80	95	100	88					
clf16	78	84	80	82	86	77	75	81	82	75	76	63	83	95	85	78				
clf17	88	75	79	75	88	80	84	84	77	72	82	60	86	79	85	75	90			
clf18	89	71	79	90	85	78	85	79	90	83	82	77	89	95	89	81	77	91		
clf19	92	89	84	91	82	75	93	79	85	84	90	81	77	88	89	84	87	86	100	
clf20	88	86	90	87	87	85	75	95	88	89	93	80	91	96	97	93	84	95	86	100

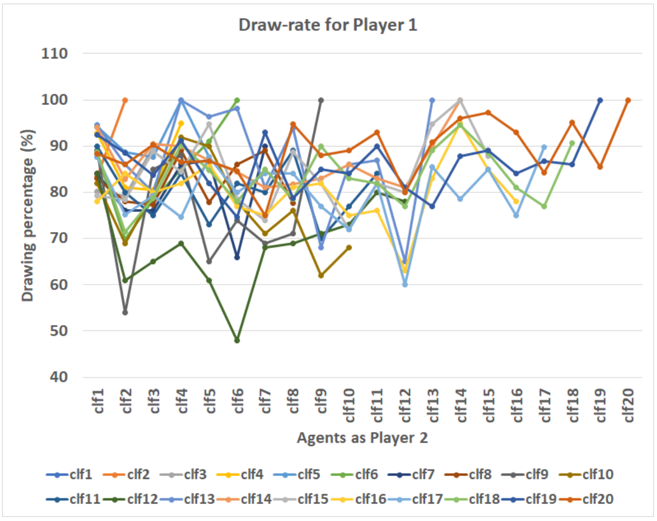


Fig. 5: Drawing-rates plotted for 20 vs. 20 agents

5 Discussions

The 20 DT-based agents play against themselves to observe any performance improvement. The outcome for each player in a game is win, draw or lose. The best result is winning, followed by a match draw. The results obtained show that the agents clf7, clf9 – clf12, and clf16 perform worse than other agents as consecutive winning peaks are observed whenever any agent plays against it. Therefore, the worst agents are discarded to avoid confusion while evaluating agents' performance.

The win-rates and draw-rates of selective agents can be seen in the table. 3 and 4, respectively. The highest values are highlighted in yellow to show the peak.

Table 3: Win-rates of selective agents

Player 2

Player 1	clf1	clf2	clf3	clf4	clf5	clf6	clf8	clf13	clf14	clf15	clf17	clf18	clf19	clf20
clf1	7.5													
clf2	4.8	0												
clf3	14	10	6.3											
clf4	7.2	4.2	6.8	3.2										
clf5	5.8	5.3	7.5	0	7.1									
clf6	12	17	15	8.5	4.5	0								
clf8	11	11	15	6.4	5.5	7.4	12							
clf13	5.3	12	12	0	0.5	0	4.1	0						
clf14	6	10	0	3.4	0	3.3	4.6	0.9	0					
clf15	10	17	5.7	2.4	0	2.8	5	0	0	7.1				
clf17	7.6	5.6	12	13	5.8	10	8.3	3.4	13.3	9.2	5.8			
clf18	11	13	5.5	6.8	1.2	9.4	12	6.9	5.4	6.4	7.4	5.7		
clf19	2.4	6	1.7	3.9	12	5.8	3.7	13.8	8.3	6.6	4.3	5.9	0	
clf20	12	0	0	8.7	6.4	4.8	5.3	2	2	0	6.2	0	7.8	0

Table 4: Draw-rates of selective agents

Player 2

Player 1	clf1	clf2	clf3	clf4	clf5	clf6	clf8	clf13	clf14	clf15	clf17	clf18	clf19	clf20
clf1	88													
clf2	83	100												
clf3	80	79	90											
clf4	93	81	81	95										
clf5	94	89	88	100	88									
clf6	88	70	78	85	91	100								
clf8	83	78	77	89	78	86	78							
clf13	95	86	80	100	96	98	94	100						
clf14	94	83	91	90	87	85	82	90	100					
clf15	79	78	89	84	95	79	89	95	100	88				
clf17	88	75	79	75	88	80	84	86	79	85	90			
clf18	89	71	79	90	85	78	79	89	95	89	77	91		
clf19	92	89	84	91	82	75	79	77	88	89	87	86	100	
clf20	88	86	90	87	87	85	95	91	96	97	84	95	86	100

The ideal behavior of the agents must be a gradual decrease in the win-rates curve for player 1, showing that the new agents perform better than the previous agents. For the draw-rates, the curve must ideally have a gradual increase showing that the game gets harder when the agents become smarter. The selective agents' win-rate and draw-rate curves are plotted in fig. 6 and 7, respectively.

The desired decreasing curve is observed in the winning rate of the agents as most of the peaks are observed when the future agents play with its initial versions. However, after the training of clf5, the agents start to gradually reach another peak then continue exhibiting the same behavior, as if the learning was initiated from the beginning. Furthermore, any significant improvement in the winning-rate is not observed after clf5.

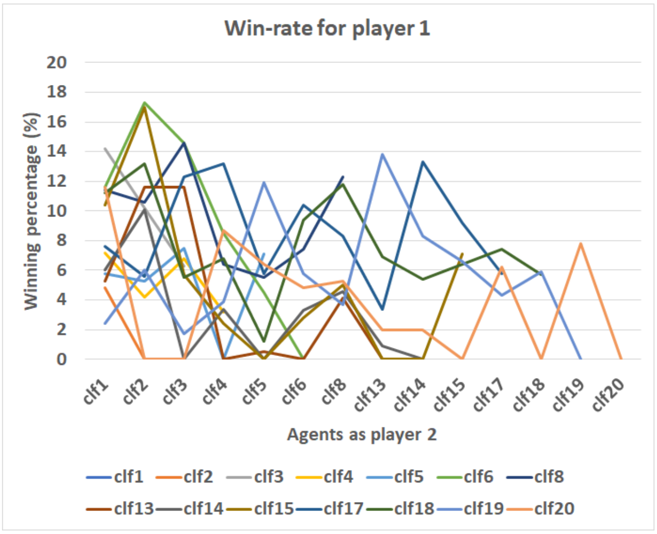


Fig. 6: Win-rates plotted for selective agents

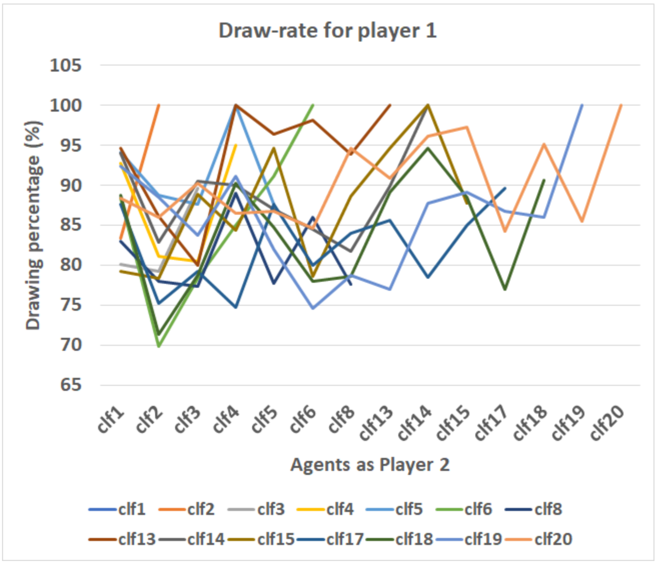


Fig. 7: Draw-rates plotted for selective agents

On the other hand, the drawing-rates form a gradually increasing curve. Most of the peaks are observed when the agents play against itself, showing that the agent has learned well. A consistent behavior observed is that after every three learning iterations of agents, the draw-rate curve drops significantly and gradually starts to attain a peak for another three iterations. The draw-rate keeps improving after each set of iterations. The first DT-based agent (clf1) has a higher draw-rate when it plays against any agent, which can be because it uses the dataset generated from UCT/MCTS for training.

6 Conclusions

In this project, MCTS is augmented with supervised learning to generate datasets. Self-playing DT-based agents

generate the dataset for 2000 iterations. The DT-based agent performs iterative learning by self-playing for 20 iterations.
In the second part, an empirical analysis based on the win-rate and draw-rate is performed to evaluate the performance loss or improvement of agents over the iterations by the 20 agents playing against its previous versions for 1000 (5x200) games.

The results indicate that to achieve a higher win-rate, five learning iterations yield the best performance under the set parameters. To achieve a high draw-rate, fourteen learning iterations provide the best results.

Due to the limitation of the system, the dataset and learning iterations are limited in this project. With a better system, the agents might indicate better performance. As this is a research project, therefore, the focus was set on both win-rate and draw-rate for evaluation. However, a single evaluation metric can be targeted to achieve a high rate for the agent.

REFERENCES

[1] S. Lucas, "Ms Pac-Man competition", ACM SIGEVOlution, vol. 2, no. 4, pp. 37-38, 2007. Available: 10.1145/1399962.1399969.

[2] J. Fu¨rnkranz and M. Kubat, Machines that learn to play games. Huntington: Nova Science, 2001.

[3] Yannakakis, G.N. and Togelius, J., Artificial intelligence and games (Vol. 2). New York: Springer, 2018.

[4] L. Yuxi. "Deep reinforcement learning: An overview." arXiv preprint arXiv:1701.07274, 2017.

[5] M. K., J. von Neumann and O. Morgenstern, "Theory of Games and Economic Behaviour.", Journal of the Royal Statistical Society, vol. 107, no. 34, p. 293, 1944. Available: 10.2307/2981222.

[6] D. Knuth and R. Moore, "An analysis of alpha-beta pruning", Artificial Intelligence, vol. 6, no. 4, pp. 293-326, 1975. Available: 10.1016/0004-3702(75)90019-3.

[7] J. Nijssen, "Monte-Carlo Tree Search for Multi-Player Games", Ph. D, SIKS, the Dutch Research School for Information and Knowledge Systems, 2013.

[8] Coulom, R. Efficient selectivity and backup operators in Monte-Carlo tree search. In 5th Int. Conf. Computers and Games (eds Ciancarini, P. & van den Herik, H. J.) 72–83 (2006).

[9] Kocsis, L. & Szepesvári, C. Bandit based Monte-Carlo planning. In 15th Eu. Conf. Mach. Learn. 282–293 (2006).

[10] Browne, C. et al. A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games 4, 1–49 (2012).

[11] D. Silver et al., "Mastering the game of Go without human knowledge", Nature, vol. 550, no. 7676, pp. 354-359, 2017. Available: 10.1038/nature24270.

[12] Świechowski, M., Tajmajer, T., & Janusz, A. , "Improving hearthstone ai by combining mcts and supervised learning algorithms.", In 2018 IEEE Conference on Computational Intelligence and Games (CIG) (pp. 1-8). IEEE, (2018).

[13] P. Cowling, E. Powley and D. Whitehouse, "UCT Monte Carlo Tree Search algorithm", www.mcts.ai, 2012. [Online]. Available: https://drive.google.com/file/d/1aeBVSFMlwVUqqnqta3wU-5tG5iiV-kpI8/view. [Accessed: 18- Apr- 2020].

[14] Browne, C. et al. A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intell. AI Games 4, 1–49 (2012).

[15] Lucas, "Ms Pac-Man competition", ACM SIGEVOlution, vol. 2, no. 4, pp. 37-38, 2007.