Durak, Don't Be the Fool: A Monte Carlo Tree Search Approach

ELIZABETH LIPIN AND BRIAN TANG AND STEPHENIE WORTHY

ACM Reference Format:
Elizabeth Lipin and Brian Tang and Stephenie Worthy. 2023. Durak, Don't Be the Fool: A Monte Carlo Tree Search Approach. 1, 1 (January 2023), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PROBLEM DESCRIPTION

Card games are frequently played around the world. Ranging from Uno to Solitaire to Poker to Euchre, there exists a multitude of games to choose from. However, oftentimes it can be difficult to find enough people to play multiplayer games with. By leveraging AI-based search algorithms, substituting human players with AI players has become ever easier. Now, all one has to do is find the desired game online and the game can be played against computer agents. Algorithms to improve the gameplay of these agents are still in the works. For this project, we will be focusing on expanding a particular algorithm, Monte Carlo Tree Search [2], to work with the card game Durak [4]. Durak is a Russian card game with complex gameplay which can be difficult to master. We aim to use Monte Carlo Tree Search (MCTS) to create an AI capable of playing on a level comparable to human players.

1.1 Team Contributions

Elizabeth Lipin primarily focused on establishing the environment for Durak. She ensured that the game properly executed turn progression, found playable cards, displayed the gameplay, and incorporated/followed the game rules. Elizabeth also assisted with the MCTS implementation. Specifically, she focused on the expansion and simulations step and the multiple card strategy implementation. She also assisted with the back propagation step and the selection step.

Stephenie Worthy primarily focused on understanding MCTS role within Durak and implementing the MCTS in Durak. Specifically, she focused on understanding the MCTS algorithm, how it is applied and how it would need to be adapted to work with Durak. Stephenie was responsible for implementing the overall MCTS algorithm loop, importing and building the tree, the selection step, the back propagation step, and the high card and low card strategies. Additionally, she assisted with implementing the expansion and simulation steps.

Brian Tang primarily focused on ranking the playable cards and evaluating the MCTS algorithm within the Durak environment. Specifically, he devised the metrics to evaluate the algorithm. Additionally, he assisted with the implementation of the MCTS algorithm. Brian was responsible for implementing the passing strategy and ensuring it aligned with gameplay rules. He determined an adequate metric to determine wins for the passing function since passing is typically considered a loss if not done voluntarily.

Author's address: Elizabeth Lipin and Brian Tang and Stephenie Worthy.

Manuscript submitted to ACM

2 RELATED WORK

Several works have explored using MCTS for solving multiplayer card games. For example, research has been done to produce AI capable of determining solvable game states and finding optimal moves for Birds of a Feather Solitaire [5]. Another work from Choe *et al.*also explored using MCTS for card games – in this case, a digital card playing game, Hearthstone [3]. The game we plan to apply MCTS to, Durak, has been shown to be computationally complex for multi-player games. Durak, being a 2-6 player game, serves as a challenging problem space for evaluating search algorithms designed to find optimal moves/responses. For example, Bonnet *et al.*proved that finding an optimal move against an attack in Durak is NP-Hard [1]. MCTS has been shown to excel for games with hidden information and randomness [2, 6].

MCTS traverses the game tree from root to end using a specific strategy and the Upper Confidence Bound (UCB) formula. This formula is crafted to balance the trade-off between exploration and exploitation. S_i is a value of node i, x_i is the mean of node i, C is a constant, and t is the total number of traversals/simulations. The MCTS algorithm consists of expansion, simulation, and back propagation.

$$S_i = x_i + C\sqrt{\frac{\ln(t)}{n_i}}$$

Durak traditionally utilizes a 36-deck hand, though we use 52 cards, attacking up to 6 pairs, and clockwise attack rotation. If the defender wins, the cards are discarded, otherwise, they must pick up all the cards played. Durak is a complicated game and few online versions can be found. Of these versions, several we tried were not very good, with computer players making many illogical and strategically unsound plays. Most of these games use force play, despite that not being a true rule of the game. Our version allows players to pass, provided they are not making the first attack. This is consistent with standard gameplay rules.

Overall, our project introduces a common approach to learning with an uncommonly executed game. Implementing the Monte-Carlo Tree Search learning algorithm improves upon the online versions of Durak we found.

3 METHODOLOGY

3.1 Datasets and Environments

3.1.1 Environment Setup. This particular problem involves interactions in the environment of the card game Durak. The parts of this environment include the cards and the players. This is a multi-agent game and it tends to be stochastic, as the next state depends both on the current action and the random cards received at the end of the turn. The environment is static, as it cannot change while the agent is making a decision on which cards to play. It is discrete as there is a finite number of states. When it comes to the number of states, the answer is complicated. At a minimum, the number of states is 387, which is the total number of pairs possible. These pairs would be grouped in up to 6 pairs per turn with the combinations calculated at 1,251,677,700 if all 6 possibilities are used. The types of pairs which can be grouped together are chosen by card numbers, so this would actually end up far smaller. For the sets of 2 cards mentioned earlier, the number of combinations was split almost in two, from 630 combinations to 387. It is expected that the combinations involving more cards will be cut even more. The overall maximum, without the removals given by the rules, comes out at 2,241,812,647. Again, this is much higher than the actual maximum, with rules included, but that ended up being a bit too complex to work through.

The base input structure is the starting player gives the next player a card. The next player then covers the card with a card of higher value. One of the suits is called the trump and has a higher value than cards of other suits (ex: a 4 of the trump suit is higher than a king of a non-trump suit). Players can add up to six cards for the player to cover. The output

is the covering player either beats or takes all cards and all players take enough cards to have six in their hands again. This specific structure can vary slightly in various versions of the game.

We have completed a python terminal version of Durak with AI capable of selecting random cards to play. Our development of the game involved coding cards, rules, setup, players, and attacking/defending cycles. Users can select a number of players between 2 and 4 (though in a realistic game, any number of players is realistically possible). Users can also choose whether or not to play themselves. If the user chooses to play, they take on the role of player 0, and all other players' cards are hidden from the user. If the user does not play, all cards are open. The terminal highlights playable cards in red, shows when the final dealable card, the trump card at the center , is taken (by highlighting it in blue), and shows all of the cards played within a turn. At the end, it informs the user who lost the game, or in other words, who ended up as the Durak (fool). You can find our Python implementation of Durak at https://github.com/Lisa-Lipin/DurakAI

Fig. 1. A screenshot of our Durak game simulation environment with only computer players.

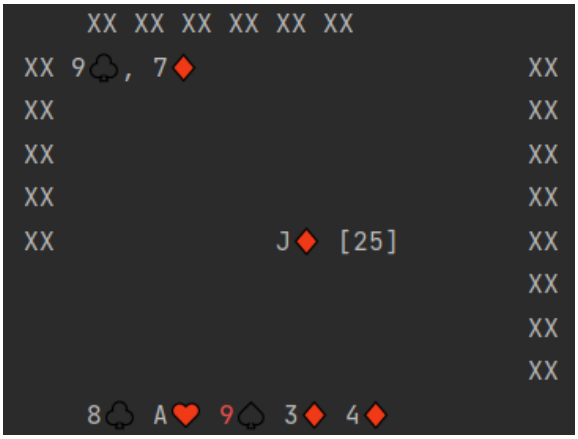

Fig. 2. A screenshot of our Durak game simulation environment with only partial information provided for the human player.

3.1.2 State Representation. As we work under the assumption of a partially observable environment, representing the exact cards in the AI's hand becomes less useful due to the randomness of other players' hands. We utilize a heuristic based on whether the card is the trump suit ($t = 0, 1$), the set of previously played cards matching the suit (p_s), the set

of previously played cards matching the rank (p_r), the card sets/pairs in the hand (r_h), the card rank ($r_c = 2, ..., 14$), and weight for the value of the trump ($c \geq 0$). This heuristic is computed for each card in the players' hands and is given in Eq. (1). We represent the state of the game as a vector containing the number of cards played, the defending player, and the heuristic values of each card in the player's hand (Eq. (2)).

$$H(x) = r_c + (t \cdot c) + |r_h| - |p_s| - |p_r| \tag{1}$$

$$State = [p, d, h_1, h_2, h_3, h_4, h_5, h_6] \tag{2}$$

3.1.3 Policy Selection. Ultimately the ideal strategy in Durak is to discard lower-valued cards and collect higher-valued cards and trumps. However, the optimal strategy shifts to play more aggressively when few or no cards remain in the deck to draw from. The initial attacker in Durak must select a card to play based on this state representation. There are usually 6 possible options, though this can change as play continues. Subsequent attackers can either continue attacking or pass. The defender must decide whether to defend or pass (and possibly take many cards) in each round. To reduce the search space for MCTS policy selection, we threshold the heuristic values in the state to 4 ranges of values. Each playable card in the hand is represented by one of these values. After a single round is completed, the win ratio is determined by computing the heuristic for the player's hand. Wins occur for the AI player whenever it successfully defends an attack or attacks and obtains a higher average hand value. Half wins occur when the player passes on a subsequent attack. Losses occur when the player loses while defending or attacks and draws a lower average-valued hand. For multiple card strategies, whenever there is a choice between playing cards within sets, there is a random selection with weight placed on non-trump cards. We also added the option for our agents to pass as both a non-initial attacker and a defender. Passing was rewarded whenever the agent's hand contained high-valued cards and there were still cards left in the deck to draw. Passing as a defender was rarely rewarded.

3.2 MCTS Model

As Durak is a multiplayer game with a large branching factor, we needed to implement a Monte-Carlo Tree Search algorithm for the card selection of computer players. This was done by using a strategy model for the tree. The computer has the option to play either the lowest-ranked playable card in their hand, the highest-ranked playable card in their hand, playing one of the multiple cards of the same rank, or passing and picking up all cards played in that round. There are separate trees for the defender and the attacker. Both the attacker and defender trees are shared by all players. This enables the model to improve faster than each computer player having individual trees that can only be updated on their turns. The algorithm takes in the current player's hand, the trump card, current cards that are in play, and whether the current player is an attacker or defender. It first loads the corresponding attacker or defender tree for the current player and sorts the nodes in topological order as seen in Fig. 3. The second row represents the current player's strategy options, the third row represents the opposing player's strategy options, and alternates between the players until the leaf node is reached. Each node contains a numbered name associated with the node, a list of children nodes, a parent node, a strategy detailing highest or lowest, a player detailing it as an attacker or defender, the number of simulations done on that node, and the number of wins generated for that node. A random hand is generated to simulate a potential hand for the opposing player, with the same initial number of cards as this opposing player, since the current player would not be aware of the cards actually in the opposing player's hand. The MCTS algorithm updates a tree and selects a strategy by performing a selection step, expansion step, simulation, and back propagation for a set number of

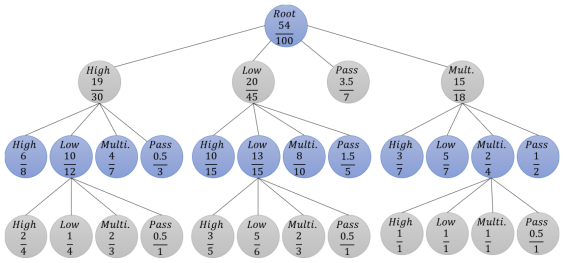

Fig. 3. Example of Monte Carlo Search Tree

iterations to determine the average utility of a given state. In order to keep the tree from becoming too large too quickly, the number of iterations was limited to ten. Once all iterations are performed, the move with the highest number of playouts is returned as the suggested strategy for that player. Each time a player has a choice between multiple cards to play, the MCTS is implemented to determine the play for that player. In the selection step, the algorithm chooses nodes starting from the root node until it reaches a leaf node by selecting the successor node with the highest UCB1 score. The UCB1 is calculated using the number of wins during past simulations, U(n), the number of simulations applied to that node, N(n), and a constant, C, to balance the exploitation and exploration terms. The constant was set to 2.5 to optimize the balance between the exploration and exploitation of the strategies.

$$UCB1(n) = \frac{U(n)}{N(n)} + C\sqrt{\frac{logN(Parent(n))}{N(n)}} \tag{3}$$

If there is a tie for the UCB1 score between two nodes, a node is selected at random with uniform probability. The nodes selected are stored within an array to make up the strategy. Four new child nodes are generated from the last selected leaf node in the expansion step to represent each possible strategy. All nodes initially have zero wins and zero simulations. If the last selected leaf node strategy is to pass, the expansion step is skipped. Nodes are not expanded in this case because once a player chooses to pass the play is over. From here, a playout is simulated based on the strategy determined during the selection process. Moves are chosen for the current player and the opposing player according to the strategy chosen. If the simulation makes it to the new nodes and the player associated with those nodes, either defender or attacker, is able to play a card, that results in a win for that player. If that player does not have a playable card, that results in a loss for that player. If a multiple strategy was selected, and there were no cards of equal rank in the hand, the simulation would re-select from that node. This selection process would repeat on every consecutive round of the simulation, ensuring an appropriate path was followed from the initial replacement node. If one of the strategies chosen is to pass, the simulation ends at that node because the play would end. The nodes within the chosen strategy are then updated based on the results of the simulation in the back propagation step. For a win, all nodes receive an increment by one for the wins and the simulations. The nodes within the strategy that align with the player associated with the leaf nodes are incremented by one for the wins and the simulations as well. The nodes aligned with the losing player within the strategy would then only have the simulations incremented by one. If during the simulation one of the players does not have a playable card before the leaf nodes are reached the associated player is considered the loser for that round. The back propagation step is then initiated from that node up to the root node with subsequent nodes remaining unchanged. The nodes that were reached and associated with the winning player have their wins and simulations increased by one. For the nodes that were reached and associated with the losing player,

their simulations increased by one. In this case, the leaf nodes previously generated in the expansion step would not have their simulations updated and would remain zero since they would not have experienced any simulations. These leaf nodes are then deleted since they were not reached. This ensures we do not have an issue in calculating the UCB1 score for subsequent runs. As previously stated, these four steps are then repeated 10 times to update the tree. Once all iterations are performed, the file containing the tree corresponding to the current player is updated to reflect the simulations performed. The move with the highest number of playouts is returned as the suggested strategy for the current player. A card to play is selected based on this strategy.

4 EXPERIMENTS

4.1 Game Simulations

For our evaluation, we played several games together to run simulations and collect data with our MCTS. We evaluated it against a uniformly random policy selection. We let one computer player use the MCTS algorithm while the other three players used a simple random selection policy for attacking and defending. We recorded the number of hand wins, game wins, and strategy selections. Our results report 200 games for each constant value varied.

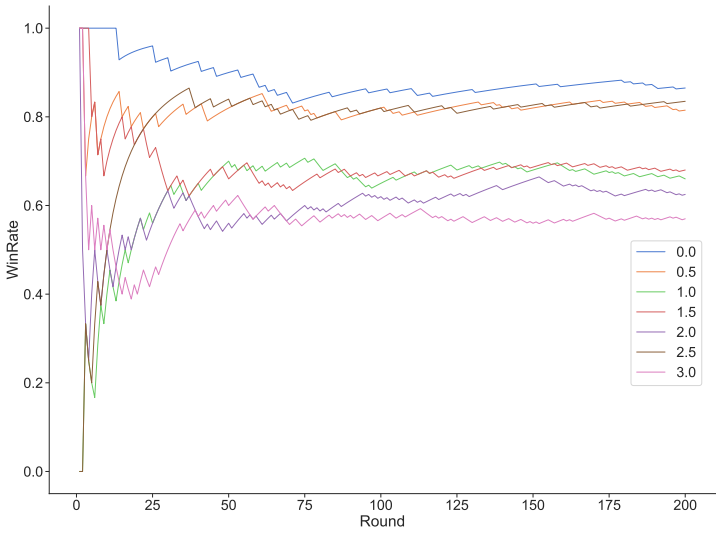

Fig. 4. The win rate of a player using MCTS in a 4-player game.

4.2 Evaluation

The model's efficiency was evaluated using the percentage of games won for a number of random initial game states. We compared the performance of our MCTS implementation with a random card selection policy for 4-player games. The results from varying the constant value in our UCB1 algorithm are presented in Fig. 4, which demonstrates that our method on average achieves a higher win rate compared to a purely random selection policy. The highest win rate is 84.07% for a 4-player game, occurring for the exploitation constant of 0. Other high win rates occurred for constants of 0.5 and 2.5. The larger constant terms for the UCB1 algorithm perform worse. We also record the number of turns and

their respective card selection policies to gain insight into how MCTS informs card playing. Generally, opting to play the lowest-valued playable card in the player's hand leads to more game losses. Intuitively, this is because, towards the end of the game, players need to play more aggressively to avoid losing. The discrepancies in the game wins between the selected constants likely results from our complex heuristics (the selection policy and win assignment conditions).

Figure 6 portrays the win rates conditioned on the ratio of highest to lowest card selection. Finally, Fig. 5 indicates that our MCTS approach wins almost all hands and that hand wins are correlated with game wins. This means our approach of using hand wins and other metrics to calculate game tree wins is a suitable substitute for calculating entire game wins.

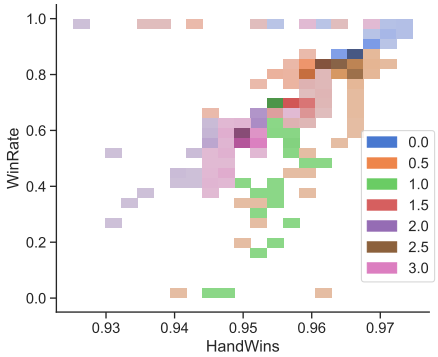

Fig. 5. The win rate of a player using MCTS in a 4-player game conditioned on the percentage of hand wins.

Overall, our results suggest three things: (1) our MCTS algorithm is able to outperform random play, (2) our MCTS method calculating hand wins is suitable for approximating game wins, and (3) an MCTS model with access to the full game state would be more optimal than the strategy selection MCTS.

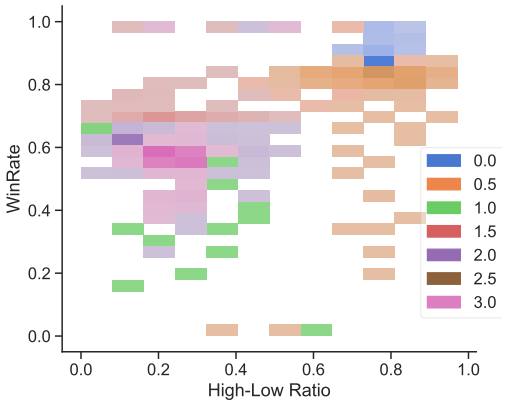

Fig. 6. The win rate for various strategies and constant values.

4.3 Limitations

Unfortunately, the time required to complete a game search is considerable due to a large number of iterations and propagations MCTS algorithms generally require. Currently, most of the runtime is spent on simulating future game nodes and the back propagations. Another limitation of our approach is that even though higher amounts of hand wins were correlated with higher game win rates, it does not necessarily imply that the agent will avoid becoming the Durak. The random nature of Durak also results in often unpredictable circumstances where playing optimally with limited knowledge still results in a loss.

5 POSSIBLE FUTURE WORK

One of the biggest things we could add is alternate versions of the game. To the nearest translation, passing, add-on, and circular Durak are all traditional versions of the game and yet these versions aren't typically found online. The next likely step would be adding the ability to play these versions and the option to choose a version.

Also, we were unable to add levels to the game as of yet. Adding the ability to play on easy mode or hard mode would greatly improve the game for human players.

REFERENCES

[1] Bonnet, É. The complexity of playing durak. In *25th International Joint Conference on Artificial Intelligence (IJCAI 2016)* (2016), pp. 109–115.

[2] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games 4*, 1 (2012), 1–43.

[3] Choe, J. S. B., and Kim, J.-K. Enhancing monte carlo tree search for playing hearthstone. In *2019 IEEE Conference on Games (CoG)* (2019), IEEE, pp. 1–7.

[4] Contributors to Wikimedia projects. Durak - Wikipedia, Aug. 2022. [Online; accessed 29. Sep. 2022].

[5] Roberson, C., and Sperduto, K. A monte carlo tree search player for birds of a feather solitaire. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 9700–9705.

[6] Whitehouse, D. *Monte Carlo tree search for games with hidden information and uncertainty.* PhD thesis, University of York, 2014.