Online Planning Strategies for Gin Rummy

Anthony Vento
Department of Electrical Engineering
Stanford University
Stanford, CA, USA
avento@stanford.edu

Shafat Rahman
Department of Computer Science
Stanford University
Stanford, CA, USA
shafatr@stanford.edu

Abstract—**In this work we explore strategies for determining optimal actions in Gin Rummy, a turn-based multi-agent card game. We focus only on online planning algorithms since Gin Rummy has a high-dimensional state space that is computationally intractable. We first present playing Gin Rummy as a Markov Decision Process (MDP) and then implement four online decision-making algorithms: lookahead with rollouts, forward search, sparse sampling, and a variation of Monte Carlo tree search. The algorithms make different assumptions about the knowledge of the current state of the environment that is available to the agent. We use a random agent and a greedy agent as benchmarks to evaluate the time and decision-making efficiency of the four online algorithms. We also simulate rounds where the agents following the different online strategies compete against each other. We find that forward search with depth 1 and sparse sampling with depth 1 and 10 samples outperform the other algorithms in making decisions efficiently.**

Index Terms—**Markov Decision Process, Multi-agent, Online Planning, MCTS**

I. Introduction

Gin Rummy is a popular card game where two players compete to form sets and runs using the cards in their hand in as few turns as possible. Gin Rummy has been around since 1909 and has continued to gain worldwide popularity. Some claim that it is "one of the most widely spread two player card games of all time." Much of this popularity can be attributed to the many different decisions a player needs to take at each turn. These include decisions regarding drawing cards, forming good hands, and choosing the right time to challenge the opponent in order to win the game. Since there are a lot of complicated decisions involved, we are interested in exploring decision-making algorithms that can help choose optimal decisions at each turn. In particular, we aim to develop a player that can defeat both a random player and a greedy player consistently. A random player can be thought of as a novice who is still learning the game, while a greedy player represents a myopic intermediate player.

II. Gin Rummy Rules

A. Setup and Goal

Gin Rummy is a two-player card game where each player is dealt 10 cards from a standard deck of 52. A player's goal is to form melds with the 10 cards in their hand while minimizing their deadwood count.

B. Melds

Players can form two types of melds: sets and runs. A *set* is a combination of 3 or 4 cards of the same rank. A *run* is a combination of 3 or more consecutive cards of the same suit. One card cannot be used to form multiple melds at the same time. A *deadwood card* is a card in a player's hand that is not in a meld. *Deadwood count* is the cumulative rank of the deadwood cards in a player's hand. In Gin Rummy the value of a Jack, Queen, and King is ten, the value of an Ace is one, and the value of all other cards is the rank of the card.

C. Turns

After each player has been dealt 10 cards, one card from the deck is flipped up to start the discard pile. During each turn, a player must pick up one card from either the top of the deck (disclosed) or the top of the discard pile (revealed). The rank and suit of all cards in the deck and the ordering of the deck are unknown to both players. The rank and suit all cards in the discard pile are known to the players. After drawing a card the player must discard one card from their hand and place it flipped up at the top of the discard pile. A player may choose to discard the card they just drew or any of the other 10 cards that they already had. Players continue taking turns until the round ends.

D. Ending a Round

To end a round, a player can call knock or gin. A player can call *knock* when the deadwood count of their hand is 10 or lower. They can call *gin* when the deadwood count of their hand is 0. Once a player has called knock or gin, both players in the game reveal their hands. The opponent (player that did not call knock or gin) is then allowed to *lay off* some of their deadwood cards by combining them with the melds of the first player. This allows the opponent to decrease their deadwood count. The player who announces knock is not allowed to lay off their deadwood cards.

E. Scoring

If player 1, announces knock and if their deadwood count is lower than the player 2's deadwood count after player 2 has laid off their deadwood cards, then player 1 wins the round and earns points worth the difference of the players' deadwood counts. However, if player 1's deadwood count is equal to or less than Player 2's deadwood count then Player 2 wins the round and earns an extra 10 points above the difference of the players' deadwood counts. If player 1 announces gin, they win the round and earn 20 points plus point worth player 2's deadwood count. Typically, a game is played over multiple rounds to 100 points.

III. Related Work

There have been several attempts to find efficient reinforcement learning algorithms for decision-making in Gin Rummy. Kotnik & Kalika compare two approaches: temporal-difference learning and co-evolution of the value function, to decide which action to perform at each turn [1]. They use self-play to gather data and train the agent. In self-play an agent learns by competing against a copy of itself.

Nagai et al. use genetic and grid search algorithms to optimize hyperparameters that drive decisions about whether to discard a card, draw a card, or call knock or gin [2].

There have also been successful attempts to develop agents that efficiently determine optimal actions in other turn-based multiaction games such as Hero Academy by using online planning algorithms such as Monte Carlo tree search [3]. However, we did not find any direct applications of online planning algorithms to decision making in Gin Rummy.

IV. Gin Rummy as a Markov Decision Process

We present each round of Gin Rummy as a Markov Decision Process (MDP), by defining the state of environment, the possible actions, the reward, and the transition model.

A. State of Environment

In a game of Gin Rummy, the current state of the environment consists of:

1) Agent 1's hand (h_1)
2) Agent 2's hand (h_2)
3) Deck of remaining cards to draw from (c)
4) Discard pile (p)
5) Agent turn (t)

Therefore, each state, s, is a function of h_1, h_2, c, p, and t.

B. Possible Actions

An agent can take one of 21 possible actions at each turn:

1) Draw from deck, discard card 1 from hand
2) Draw from deck, discard card 2 from hand

$\vdots$

10) Draw from deck, discard card 10 from hand
11) Draw from deck, discard drawn card
12) Pick up top discard card, discard card 1 from hand
13) Pick up top discard card, discard card 2 from hand

$\vdots$

21) Pick up top discard card, discard card 10 from hand

Note that we ignore the action that involves picking up the card at the top of the discard pile and then discarding it, since this action only does not change the state of the environment significantly. It changes only one variable, agent turn, and thus can be ignored.

For simplicity we also make two key assumptions: an agent greedily calls knock if they are able to and an agent will never call gin. We made this assumption because we focus on one round of Gin Rummy and not an entire game to 100 points. Therefore, an agent does not need to explicitly decide whether to call knock. They are also relieved from taking decisions such as whether to keep playing in order to be able the call gin once their deadwood count is below 10. This also means that an agent does not have to consider the opposing agent's deadwood count.

C. Reward

We assign a reward associated associated with the current state and the action, $R(s, a)$, to each agent based on their deadwood count and the number of melds in their hand:

$$R(s, a) = (4 \times \text{\# of melds}) - (5 \times \text{deadwood count}) \quad (1)$$

We add an additional 100 points to the reward if the deadwood count is less than or equal to 10. The reward follows from the rules of Gin Rummy: it penalizes high deadwood counts and rewards the agent for forming melds that minimize their deadwood count.

D. Transition Model

$T(s'|a, s)$ defines the probability of transitioning from the current state, s, to a new state, s', given that the agent takes action a. Note that the state transitions are deterministic, i.e., there is only one possibility for s' given a and complete knowledge of every component of s.

While deciding which action to take, and agent is only interested in the transition model from its point of view. That is, if we are in state s_1 (i.e. it is Agent 1's turn) then we are interested in $T(s'_1|a_1, s_1)$ where s'_1 is the next time it is Agent 1's turn and a_1 is the action Agent 1 takes.

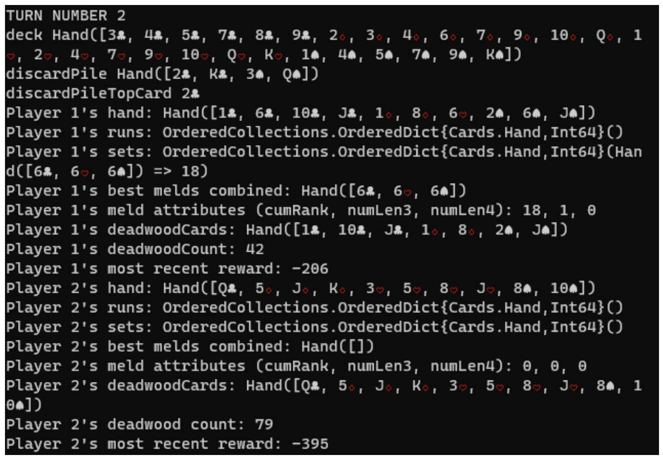
$$T(s'_1|a_1, s_1) = \sum_{a_2=1}^{21} T(s'_1|a_1, s_1, a_2) P(a_2|a_1, s_1) \quad (2)$$

From a single agent's perspective, we assume that the opposing player's actions are random. If we define a_2 to be Agent 2's action, we have

$$T(s'_1|a_1, s_1) = \frac{1}{21} \sum_{a_2=1}^{21} T(s'_1|a_1, s_1, a_2) \quad (3)$$

Again, we note that $T(s'_1|a_1, s_1, a_2)$ assumes that an agent has knowledge of all of s_1 including the opponent's hand and the deck.

However, in the methods we discuss, an agent never has complete knowledge of the state. For example, the agent does not know what the top card in the deck is. From the agent's perspective the top card of the deck could be any card not in the agent's hand and not in the discard pile. Therefore, in order to consider all possible next states, the agent has to simulate all possible actions for all possible cards at the top of the deck.



```
TURN NUMBER 2
deck Hand([3♣, 4♣, 5♣, 7♣, 8♣, 9♣, 2♢, 3♢, 4♢, 6♢, 7♢, 9♢, 10♢, Q♢, 1
♡, 2♡, 4♡, 7♡, 9♡, 10♡, Q♡, K♡, 1♠, 4♠, 5♠, 7♠, 9♠, K♠])
discardPile Hand([2♣, K♣, 3♠, Q♠])
discardPileTopCard 2♣
Player 1's hand: Hand([1♣, 6♣, 10♣, J♣, 1♢, 8♢, 6♡, 2♠, 6♠, J♠])
Player 1's runs: OrderedCollections.OrderedDict{Cards.Hand,Int64}()
Player 1's sets: OrderedCollections.OrderedDict{Cards.Hand,Int64}(Han
d([6♣, 6♡, 6♠]) => 18)
Player 1's best melds combined: Hand([6♣, 6♡, 6♠])
Player 1's meld attributes (cumRank, numLen3, numLen4): 18, 1, 0
Player 1's deadwoodCards: Hand([1♣, 10♣, J♣, 1♢, 8♢, 2♠, J♠])
Player 1's deadwoodCount: 42
Player 1's most recent reward: -206
Player 2's hand: Hand([Q♣, 5♢, J♢, K♢, 3♡, 5♡, 8♡, J♡, 8♠, 10♠])
Player 2's runs: OrderedCollections.OrderedDict{Cards.Hand,Int64}()
Player 2's sets: OrderedCollections.OrderedDict{Cards.Hand,Int64}()
Player 2's best melds combined: Hand([])
Player 2's meld attributes (cumRank, numLen3, numLen4): 0, 0, 0
Player 2's deadwoodCards: Hand([Q♣, 5♢, J♢, K♢, 3♡, 5♡, 8♡, J♡, 8♠, 1
0♠])
Player 2's deadwood count: 79
Player 2's most recent reward: -395
```

Fig. 1. Simulator in Action

V. Gin Rummy Simulator

We develop a Gin Rummy simulator in order to experiment with different algorithms and methods to solve Gin Rummy. Our game simulator includes two agents in an environment. The agents keep track of their own hands, melds, deadwood count, their playing strategy, and the rewards they receive at each turn. The environment keeps track of the deck, discard pile, agents, which agent's turn it is, and the winner if there is one. Each round the agents choose one of the 21 actions depending on their strategy. The agents greedily call knock once their deadwood count is less than or equal to 10. Once an agent calls knock the opposing agent lays off their deadwood cards. Subsequently, the agent with the lower deadwood score wins. If the deadwood scores are tied then the agent that called knock wins. A round can also end if there are fewer than 2 cards left in the deck.

In order to support different card actions, we extend an existing Julia programming language package called Cards.jl to support actions in a Gin Rummy game such as dealing, discarding, and picking cards from the discard pile or the deck [4]. We also add functionality to compute melds and deadwood count in a agent's hand.

We show an example output of the simulator in Figure 1. Note that the deck also has an order which is not displayed in the figure.

VI. Methods

We investigate various online planning methods to decide which action to take at each turn in a single round of Gin Rummy. Online planning methods reason about the reachable state space from the current state to choose an action. This is especially important in problems such as our MDP formulation of Gin Rummy, where the state space is quite large, and the set of reachable states is only a small subset of the set of all possible states. For instance, there are $\binom{52}{10}\binom{42}{10}(32)(31!) \approx 6.1 \times 10^{54}$ possible states from which a round can begin. As the agents take actions the state changes depending on which cards they draw and discard. Since there are many possibilities of cards that can be drawn and discarded, especially at the initial turns, the size of the state space can become intractable quickly. Online planning methods present a solution to this problem by considering a limited number of possible next states at each turn. Some online methods converge to the optimal action at a particular state, while others do not.

We focus on 4 online planning methods: lookahead with rollouts, forward search, sparse sampling, and a variation of Monte Carlo tree search.

A. Lookahead with Rollouts

In lookahead with rollouts the agent considers all 21 actions from the current state and performs m infinite horizon rollouts (i.e. until game termination) for each. Every simulated action after the first action in each rollout are random for both the agent and its opponent.

For our formulation of lookahead with rollouts we assume that an agent has complete information about the opposing agent's hand. As a result, the agent can infer which cards are in the deck. The ordering of the deck, however, remains hidden. This assumption is important to ensure that while doing rollouts, the agent picks cards from the deck and not from the opposing agent's hand, which would be a violation of the game rules and would introduce duplicate cards to the game.

The action that leads to the highest expected utility across all m rollouts is considered to be the optimal action to perform from the current state. Of course, due to the randomness in the rollouts, this is only an approximation of the actual optimal action from the current state.

B. Forward Search

In forward search the agent considers all possible transitions up until depth d in order to determine the optimal action from the current state. At each depth the agent can take 21 actions. We assume that the agent does not know the order and composition of the deck and the composition of the opponent's hand.

Since we assume that a lot of information about the current state is hidden, the agent has to account for all possible current states while deciding the possible next states. For example, for the first turn in depth 1, if the agent wants to draw a card from the deck then there are 41 different cards that could possibly be at the top of the deck. After drawing a card the agent can then discard one of the 11 cards in its hand. Hence, there are $41 * 11 = 451$ possible states that could result from drawing from deck. The agent also has to consider picking up the top card from the discard pile and discard one of the cards from its hand, which leads to 10 more possible next states. From the perspective of the agent, each of these 461 states are equally likely.

In depth 2 of forward search each of these 461 states can result is hundreds of other states next states that the agent will have to consider. Due to this exponential growth of the state space we only experiment with depth 1 and depth 2 forward search.

The optimal action is calculated by considering each of the possible state-action pairs up to depth d, calculating the associated utility for that state-action pair using the lookahead equation, and propagating the utility upwards.

In order to calculate the lookahead state-action value from a state s_1 given an action a_1 we use

$$U^{a_1}(s_1) = R(s_1, a_1) + \gamma \sum_{s'_1} T(s'_1|s_1, a_1) U^{a_1}(s'_1) \qquad (4)$$

where γ is the discount factor (we choose $\gamma = 1$) and $R(s_1, a_1)$ and $T(s'_1|s_1, a_1)$ are defined in (1) and (3), respectively. Recall that in order to calculate $T(s'_1|s_1, a_1)$, we must sum over all possible opponent actions and all possible cards at the top of the deck. In the end, we choose the action which leads to the highest expected utility

$$\pi(s_1) = \underset{a}{\operatorname{argmax}}\, U^a(s_1) \qquad (5)$$

where $\pi(s_1)$ tells us the action to take from the current state.

C. Sparse Sampling

Like forward search, our formulation of sparse sampling considers all 21 actions at each depth. However, instead of considering all possible state transitions, it creates an approximation of the unknown cards, i.e. cards in the deck and the opponent's hand, by randomly choosing m cards from the set of unknown cards with equal probability. This results in a smaller reachable state space and depending on the size of hyperparameters, decreases decision-making time at the expense of potentially returning a sub-optimal policy compared to forward search. The hyperparameters for sparse sampling include the number of cards to sample, m, and the depth to which the algorithm must run. If m exceeds the number of unknown cards remaining then we use all the unknown cards in our algorithm.

D. Modified Monte Carlo Tree Search Algorithm

We implement a modified version of *Monte Carlo tree search (MCTS)* by combining sparse sampling and lookahead with rollouts. We first iterate through each of the 21 actions and for each action simulate 10 rollouts. Based on these rollouts we determine the initial estimates of the action value function in (4).

After setting initial estimates for the action value function, we run k_{max} simulations and for each simulation we choose the action that has the maximum upper confidence bound (UCB). Initially, the UCB for each action is set to positive infinity. Once an action is chosen, we recursively call our modified Monte Carlo tree search (MCTS) algorithm up to depth d. At depth d we call sparse sampling with depth 1 and m samples to update the approximate value for the action value function. We then propagate the approximated value up to depth 0 and update the action value approximation for the action chosen at the beginning of the simulation. We also update the UCB, which is used to balance exploration and exploitation.

In the end, we pick the action which leads to the highest expected utility (highest expected action value).

VII. Results

We use a random and a greedy strategy as baseline for evaluating the performance — win % across 100 rounds and time/turn — of our online planning methods. A random agent randomly picks one of the 21 actions at each turn, hence making a random decision about which card to discard and whether to draw from the discard pile or the deck. A greedy agent picks up the top card from the discard pile if it reduces deadwood count, and if not then it blindly draws from the deck. The greedy agent discards the highest ranked deadwood card in its hand.

For lookahead with rollouts, we experiment with different number of rollouts and report the win % vs. the random and greedy agents, as well as the average time/turn in Table I. We see a general trend where, as we increase the number of rollouts, our win % increases against both agents and the time/turn increases linearly as a function of the # of rollouts.

TABLE I
Lookahead with Rollouts Results

Rollouts	Win % vs Random	Win % vs Greedy	Time/Turn (s)
1	50	21	0.0040
5	86	53	0.0180
10	98	62	0.0313
25	98	80	0.1231
50	99	77	0.2142
100	99	81	0.4104

We also try depth 1 and depth 2 forward search and report the win % vs. the random and greedy agents, and the average time/turn in Table II. Although forward search depth 1 and 2 perform similarly against the random and greedy agent, we see that forward search depth 2 takes longer per turn.

TABLE II
Forward Search Results

Depth	Win % vs Random	Win % vs Greedy	Time/Turn (s)
1	100	81	0.0152
2	98	89	49.0430

Similarly, for sparse sampling we experiment with different values for the depth and the number of samples and report the win % vs. the random and greedy agents, and the average time/turn in Table III. We see that even though sparse sampling does not explore every possible state action transition, we notice similar win %s compared to forward search. Notice that sparse sampling with depth 2 and 20 samples performs about the same as forward search with depth 2, but is faster.

TABLE III
Sparse Sampling Results

Depth	Samples	Win % vs Random	Win % vs Greedy	Time/Turn (s)
1	10	99	77	0.0031
1	20	100	84	0.0053
1	30	99	86	0.0090
2	10	100	81	2.5626
2	20	100	83	12.6698
2	30	100	92	39.7284

For our Modified MCTS algorithm we try different values for the number of samples and report the win % vs. the random and greedy agents, and the average time/turn in Table IV. We see that although the modified MCTS can consistently beat a random agent, the strategy struggles against a greedy agent. Also note that modified MCTS with depth 1 is much slower that forward search with depth 1 in addition to having a lower win %. It also performs much worse than most of the sparse sampling algorithms we ran, and performs about the same as lookahead with 10 rollouts.

TABLE IV
MODIFIED MCTS ALGORITHM RESULTS

k_{max}	Depth	Samples	Win % vs Random	Win % vs Greedy	Time/Turn (s)
25	1	10	94	61	5.0365
25	1	20	98	65	7.6091

Finally, we also compare several of our online planning methods against each other in Table V. The parameters we used to gather this data are specified in parentheses. We find that forward search and sparse sampling perform better than lookahead with rollouts. Furthermore, we see that despite the large difference in time per turn for depth 2 and depth 1 forward search, the agents behave similarly when competing against each other. Forward search with depth 1 also performs about the same as sparse sampling with 10 samples. Of course, depth 2 sparse sampling take much longer than depth 1 forward search, but time per turn for depth 1 forward search and depth 1 sparse sampling with 10 sample are comparable. Lastly, we notice that our modified MCTS algorithm performs poorly against depth 1 forward search.

TABLE V
AGENTS COMPETING AGAINST EACH OTHER

Agent 1 Style	Agent 2 Style	Agent 1 Win %
Lookahead w/Rollouts (100)	Lookahead w/Rollouts (5)	92
Lookahead w/Rollouts (100)	SparseSampling (1,10)	54
Forward Search (1)	Lookahead w/Rollouts (5)	93
Forward Search (1)	Lookahead w/Rollouts (100)	95
Forward Search (1)	Sparse Sampling (1,10)	55
Forward Search (1)	Sparse Sampling (2,10)	52
Forward Search (1)	Modified MCTS (25,1,20)	97
Forward Search (2)	Forward Search (1)	53

VIII. DISCUSSION

Each of our 4 online planning algorithms assume varying levels of knowledge of the current state of the environment. In lookahead with rollouts and the modified MCTS, we assume that the agent knows which cards are in the deck and in the opposing agent's hand. However, the agent does not know the order of the deck. In sparse sampling and forward search we assume that the agent has no information about the cards in the deck or the opponent's hand.

Surprisingly, even though lookahead with rollouts and the modified MCTS assume knowledge of the opposing player's hand, both agents perform worse than sparse sampling and forward search. This can be attributed to the randomness involved in lookahead with rollouts and modified MCTS.

We see from our results, that most strategies are able to consistently beat a random agent showing that even with a little bit of strategy, a player should be able to beat a person without knowledge about the game. However, a human player in Gin Rummy naturally tends towards a greedy style of play. We discover that most of our algorithms are able to beat this style of play most of the time.

In particular, considering both speed and win %, the algorithms that performed best are forward search depth 1 and sparse sampling depth 1, 10 samples. Therefore, if computationally feasible in a human player's mind, an optimal strategy is to consider the expected utility drawing a card from the deck versus the expected utility for drawing from the top of the discard pile to determine the best action to take.

In the future, it would be interesting to explore algorithms that have complete knowledge of the current state and make optimal decisions without sampling from the state space or running random rollouts. Due to large amount of time some algorithms took, we gather limited data for our algorithms. This work can be extended by investigating the performance of forward search, sparse sampling and the modified MCTS algorithms at depths higher than 2.

IX. CONCLUSION

We develop online planning algorithms that beat both a random and greedy agent reliably in one round of Gin Rummy. Interestingly, the algorithms we develop that do not assume knowledge of the opposing player's hand and the cards in the deck perform better than the algorithms with this assumption. Furthermore, some of our best performing algorithms can make an optimal decision in milliseconds. Even some of our least efficient algorithms are expected to be faster than human players. We show that online planning algorithms can be used to efficiently handle decision making under uncertainty for multi-agent turn-based cards games like Gin Rummy, which, when formulated as an MDP, have intractable state spaces.

ACKNOWLEDGMENTS

This work was completed for the final project for AA 228/CS 238: Decision Making Under Uncertainty at Stanford University. We would like to thank our professor, Dr. Mykel Kochenderfer, for teaching this course. We would also like to thank the course staff for giving us insights into our project.

CONTRIBUTIONS

In general, we ensured that work was split evenly between the two of us. Both of us worked together to formulate Gin Rummy as an MDP, create the simulator, and develop relevant methods. In particular, Anthony worked on writing code for the random agent, lookahead with rollouts and forward search algorithms. Shafat coded the algorithm for the greedy agent, sparse sampling, and the modified MCTS. We both split extending the Cards.jl package, collecting results, and writing this report.

References

[1] C. Kotnik and J. K. Kalita, “The significance of temporal-difference learning in self-play training td-rummy versus evo-rummy,” in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 369–375.

[2] M. Nagai *et al.*, “A highly-parameterized ensemble to play gin rummy,” *DePauw University*, October 2020.

[3] N. Justesen, T. Mahlmann, S. Risi, and J. Togelius, “Playing multiaction adversarial games: Online evolutionary planning versus tree search,” *IEEE Transactions on Games*, vol. 10, no. 3, pp. 281–291, 2018.

[4] S. Karpinski, “Cards,” June 2020. [Online]. Available: https://github.com/StefanKarpinski/Cards.jl