Solving Liar's Dice With a POMDP

Nolan Stevenson, Chase Rupprecht

1) Link to GitHub repository here.

I. Introduction

A. How to play Liar's Dice

Liar's dice is popular multiplayer dice game where players place and challenge bids. To start the game, every player rolls the same number of dice and hides them from the other players. Each player is allowed to know the dice that they rolled and the total number of dice on the table. The first player makes a bid as to what they think the quantity of a given face value of dice are on the table. For example, they might bid: "there are three fours on the table". The quantity is three of the face value four. In this game, dice with a face value of one are wild and count as any face value die. The next player in the order can choose to raise the bid or challenge the bid. In a raise, the player must make a bid that is either higher in quantity of a given face value or a higher face value that the previous bid. For example, that player might bid "four fours" or "three fives" but that player could not bid "two fours". If a player decides to challenge, all players reveal their dice and sum up the relevant dice to the bid remembering to include ones as wild. If the number of relevant dice on the table is less than the quantity of bid, the challenging player wins, and the player who made the bid loses a dice. If the number of relevant dice on the table is equal or greater than the quantity of bid, the challenger loses a dice. After a player loses a dice, the round resets, with every player re-rolling the dice in their hand. As the game goes on the number of total dice on the table and the number of dice each player has gets lower and lower. This creates a challenging game where players who preform poorly know less and less information about what is on the table as their hand shrinks. Once a player is out of dice they are removed from the game. The winner of the game is the last player with dice.

B. Strategies

Liar's Dice is a game of probability and bluffing. One strategy that is commonly used is what players refer to as the "one third rule". This rule uses statistics to determine the feasibility of a bid. The idea is that the quantity of a bid should not exceed one third of the total dice on the table. This is because the probability of rolling any number on a six sided die is $\frac{1}{6}$, plus the quantity of rolling a one, which is wild, is also $\frac{1}{6}$. In sum, the resulting probability estimate of a number is approximately $\frac{1}{3}$ of the total dice on the table. This provides a baseline estimate players can use throughout a turn to decide if another players bid is likely or unlikely.

However, more advanced players often use dice that they rolled in order to inform likelihood. For example, if a player rolled three sixes and they have four total dice, they may be willing to make a bid on the number of sixes on the table that would violate the $\frac{1}{3}$ rule. Additionally, observing how other players bid throughout a turn can be informative to the number of dice that a player has. For example, observing a player bid a unusually high number of twos could mean that the player has a lot of twos in their hand, or, they could be bluffing.

II. Background and Related Work

Through research, most Liar's Dice implementations use game theory or machine learning models. References [1] and [3] utilize counterfactual regret minimization. Counterfactual regret minimization is an algorithm that combines game theory with iterative learning and is very powerful for solving imperfect-information games, such as Liar's Dice. In the reference [1], a counterfactual regret minimization AI won 56.9% of the time vs manual players; the expected win rate for equal opponents is 50%. Although counterfactual regret minimization only had a 6.9% increase, this is a very significant number for a game with such a random nature.

References [4] and [5] utilize Q-learning to create a Liar's Dice model. The Stanford implementation of Q-learning, [4], had unsuccessful results vs a human. The Q-learning model would only beat a human player between 20% and 40% of the time. These lower than average results were attributed to oversimplification of the state space and possibly too small of a training set. However, the article in reference [5] was able to create a very successful Q-learning model for Liar's dice. This Q-learning implementation was able to achieve a fantastic win percentage of 65% vs a human player. Interestingly, this article also mentions that a Markov Decision Process (MDP) formulation of the game worked really well, performing better than an average human.

In sum, a few variations of using AI to solve Liar's Dice have been tested and implemented before, however, the research on the subject is still very raw. Different papers have different win rate percentages and all of them use different assumptions to simplify the game. So far in our research, we have yet to find a paper or example of anybody else using a Partially Observable Markov Decision Process (POMDP) to solve Liar's Dice; thus, we believe we are presenting a novel approach.

III. Problem Formulation

Liar's dice is best represented as a Partially Observable Markov Game (POMG). A POMG contains three key features: a partially observable environment, the Markov property, and game-like interactions between players. This category of problem is notoriously difficult to solve as an agent in the game has only partial information about the state and also must make actions which consider not only itself, but other players reactions to their actions. To simplify our problem, we will be treating Liar's dice as a Markov Decision Process (MDP) and a Partially Observable Markov Process (POMDP). This greatly simplifies the game by removing a lot of game theory and allows more standard approaches to be implemented to solve the game.

Even with this assumption, the state space of the game, or possible combinations, is still massive; estimated by [3] to be upwards of 294 quintillion states. For this reason, it was decided to further simplify the game such that we could explore MDP and POMDP algorithms without outrageous computation time. To simplify the game, the potential bids a player could make are limited to only number of 6s on the table. Thus, when a player decides to bid, the action simply increases the bid quantity on number of 6s by 1. This simplification drastically reduces the state and belief space that a MDP and POMDP need to consider and makes development of the algorithms discusses in this paper much simpler. However, we were careful to ensure the algorithms used utilize techniques such as sampling to ensure they would hold up in a non-simplified game of Liar's Dice.

Both MDPs and POMDPs share the same State, Action, Transition, and Reward spaces, with POMDPs containing an additional Belief space. In this problem, the state of the game was determined to include: the number of players, each players current hand, the current player's turn, the previous players turn, the current bit, and the total number of dice on the table. The State of the game of Liar's Dice is shown in the table below:

Parameter	Structure
Number or Players	Array[Int64,1]
Players Dice	Array[Array[Int64,1],1]
Current Players Turn	Integer
Previous Player's Turn	Integer
Current Bid	[Quantity, Face Value]
Total Dice	Integer

TABLE I: State of Liar's Dice

The actions that are allow for each player are to either bid or challenge:

$$A = bid, challenge \tag{1}$$

A action of bid simply increases the current bid quantity by 1, with our simplification the face value of the bid will always be 6.

Given that the game is deterministic, the transition probability, T, of the models, given an action, is always 1. If a model decides to bid, the bid always increases the quantity by 1.

$$T_{(:)} = 1 \tag{2}$$

There was no reward function or discount factor used for this game. We did not implement a reward function because the goal of the game is simple: don't lose dice. With this idea we instead designed heuristic policies for both the MDP and POMDP model such that they attempt to not lose dice. There is no discount factor because every turn a model takes is equally as important as the last, and each turn has a finite horizon; once a player challenges, the turn ends.

In our formulation there are a few core helper function that need to be discussed. First, the *roll dice* function simply takes in the number of dice to roll and returns a vector of random numbers of that length between one and six. The *make bid* function takes in the state of the game and increases the bid quantity by one; it then updates and returns the new state. The *challenge* function takes in the state of the game and compares the hands of all of the players to the current bid. It then returns if a challenge is successful or not. A *lose dice* function resets the turn by taking in a state and player index and returns a new initialized state with that player having lost a dice.

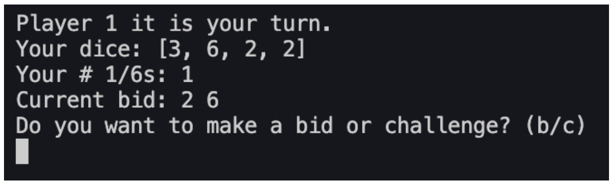
Since both the POMDP and MDP model use the same helper functions and have the same transition probability and action space, our design for simulating and benchmarking a game of Liar's Dice is very elegant. Using the program language Julia and the POMDP package [2], a MDP, POMDP, and manual player were made as POMDP data structures. Then, the following while loop can be ran to simulate a game, using inheritance programming:

Algorithm 1 Playing Liar's Dice

```
1: function PLAY(state::LDState, players::ArrayT)
2: where T<::POMDP
3:     while numPlayers $> 1$ do
4:         player $\leftarrow$ players[state.turn]
5:         action $\leftarrow$ POMDPs.action(player, state)
6:         if action == bid then                    ▷ Bid Made
7:             state $\leftarrow$ makeBid(state)
8:         else                                     ▷ Challenge Made
9:             loser $\leftarrow$ challenge(state)
10:            state $\leftarrow$ loseDice(state, loser)
11:        end if
12:        for POMDPs in players do                 ▷ Update Belief
13:            Belief $\leftarrow$ POMDPs.update(state, bid)
14:        end for
15:        state.prevTurn $\leftarrow$ state.turn
16:        state.turn $\leftarrow$ Next players index
17:    end while
18: end function
```

This function runs a while loop until there is only 1 player left in the game, the winner. At each iteration, the current player gets selected based on the current turn in the game. Then, because all models/manual players are apart of the POMDP data structure, an action is determined based on that players POMDPs.action policy. If the action is to bid, the *make bid* function is called, increasing the bid in the state. If the action is to challenge, the *challenge* function is called to determine the loser of the challenge. That player then loses a dice with the *lose dice* function, which decreases that players number of dice and resets the state, re-rolling all players dice. After a bid has been made, all POMDP models get a belief update to their belief space, based on the observation of the current turn player increasing the bid to a certain amount. Finally, after all actions and beliefs have been updated, the turn gets incremented and the while loop resets, starting the process all over again.

This implementation allows easy testing and integration between multiple models and players. Also, this implementation allowed a manual player to play with a MDP or POMDP model. The manual player's code was very easy. For the POMDPs.action function, it simply displayed relevant information to the user and asked through a terminal prompt if the user wants to bet or challenge:



```
Player 1 it is your turn.
Your dice: [3, 6, 2, 2]
Your # 1/6s: 1
Current bid: 2 6
Do you want to make a bid or challenge? (b/c)
```

Fig. 1: Terminal output to manual player

IV. Solution Approach

A. MDP Model

The first computer model that we implemented in Liar's Dice was a Markov Decision Process (MDP) model. This model is the most basic we will consider in this project. The core of our MDP is a heuristic policy that takes in information about the current state of the game and outputs which action the MDP model should take, to bid or challenge. The MDP always takes actions according to the policy. The heuristic policy that we chose to implement for the MDP was based on the one third rule strategy mentioned in the introduction. This is needed because the MDP has no knowledge of belief of what the other dice on the table could be, the MDP only knows what dice it has in its hand. The policy works as follows:

Algorithm 2 MDP Policy - One Third Rule

```
1:  function ACTIONMDP(state)
2:      yourDice ← Current 1/6s MDP has
3:      totalDice ← Total number of dice in the game
4:      otherDice ← totalDice - yourDice
5:      estimate ← yourDice + $\frac{\text{otherDice}}{3}$
6:      if estimate $<$ state.bid then
7:          return challenge
8:      else
9:          return bid
10:     end if
11: end function
```

The policy assumes the other dice on the table follow the one third rule for the number of 1/6s the other players have. The policy takes the estimate for the other 1/6s on the table and adds its own knowledge to create a net estimate for the total number of 1/6s on the table. Then, if the current bid of the game is higher than this estimate, the MDP should challenge the bid, otherwise, the MDP increases the bid.

This basic policy does a good job of replicating the strategy typically used by humans when they play liar's dice. However, only following the one third rule is very basic and doesn't take into account other players' actions and strategies throughout the game.

B. POMDP Model

To create a smarter computer model, we implemented a Partially Observable Markov Decision Process (POMDP) model. A POMDP model is typically used when the environment is partially observable, in our game, other players' dice are not observable, only your dice. However, with a POMDP, the model is able to gain information about the other players' dice based on observations seen throughout the game. The observations, o, throughout a round are the actions taken by the other players in the game; the other players increasing the bid amount. Thus, when it is not the POMDP model's turn, the POMDP model observes the action of the current player.

1) Belief State: We define the belief state of the POMDP model to be the following:

belief = Array[SparseCat[Array[Int64], Array[Float64]]]

The belief is an array of distributions, one distribution for each player. Each distribution contains integer possibilities $0 : numDice$ which represents the possible number of 1/6s that the player could have. With each of these integers, is an associated float probability of that player having that many 1/6s. In sum, this belief represents the estimate of every other player's 1/6s amount. This belief contains all of the information necessary for the POMDP to decide which action to take once it reaches its turn.

2) Belief Update: To initialize the belief at the start of each round, we use the following algorithm:

Algorithm 3 Initial Belief Distribution

```
1:  function INITIALBELIEF(numPlayers, numDice)
2:      belief ← Empty
3:      numSamples ← 100,000
4:      for player in numPlayers do
5:          counts ← Empty counting array
6:          for numSamples do
7:              rollDice ← Randomly roll numDice
8:              num1/6s ← Count number of 1/6s in roll
9:              counts[num1/6s] += 1
10:         end for
11:         belief[player] ← normalize(counts)
12:     end for
13:     return belief
14: end function
```

This function initializes the belief state based on the current number of players in the game and their associated number of dice. It initializes the belief by sampling and then normalizing the player rolling their dice and counting the number of 1/6s they rolled; the result is an accurate initial distribution of their dice.

To update the belief, given an observation of another player biding, we implemented a rejection particle filter. Our rejection particle filter begins by first random sampling dice rolls for that player. For each dice roll, we simulate the action the player would take if they had that hand. We do this by assuming the policy that the player follows is the MDP one third rule, as shown above in Algorithm 1. If the resulting action is to challenge, we reject that particle (potential hand) as the observation we observed, the player bidding, does not align with the simulated action for that particle, challenging.

Through this method, we can prune out any potential hands that player may have that wouldn't align with their decision to bid, according to our estimated policy they are following. With the resulting particles that remain, we can do a standard belief update using the equation:

$$b'[s] += Z(o|a, s') * T(s'|s, a) * b[s] \tag{3}$$

The $Z(o|a, s')$ term represents the likelihood that an observation is correct, given that you always know when another playing increases a bid, this is always 1. The $T(s'|s, a)$ term represents the likelihood of a player rolling a given number of 1/6s and can be calculated throughout the particle filter by getting the probability for each possible state. It is important to mention again that in our implementation we are only considering bidding 1/6s, so the states to keep track of for belief are only the number of 1/6s that a player has, as that would be their deciding factor on if they should bid or challenge. The following algorithm shows the full belief update process:

Algorithm 4 POMDP Belief Update

```
1:  function UPDATEBELIEF(state, bid, o)
2:      currNumDice ← player[o]'s number of dice
3:      totalDice ← Total number of dice in the game
4:      otherDice ← totalDice - currNumDice
5:      numParticles ← 100,000
6:      transProb ← Empty counting array
7:      for numParticles do
8:          rollDice ← Randomly roll currNumDice
9:          num1/6s ← Count number of 1/6s in roll
10:         estimate ← num1/6s + $\frac{\text{otherDice}}{3}$          ▷ 1/3 rule
11:         if estimate $<$ state.bid then
12:             continue        ▷ a = challenge, Reject particle
13:         else
14:             transProb[num1/6s] += 1     ▷ Accept particle
15:         end if
16:     end for
17:     belief[o] += normalize(transProb) * belief[o]
18:     return normalize(belief)
19: end function
```

Where it can be seen in the main for loop, the rejection particle filter simulates a player following the one third rule and determines if the particle should be accepted or rejected based on the outcome of the one third rule. The particle is only accepted if, by the one third rule, the best action to take would be to bid, as observed. After generating a transition probability matrix throughout the particle filter, the final step is to update the belief using Equation 3.

3) Policy: When it is the POMDP's turn to take an action, it knows its own dice and an approximate belief of the number of 1/6s each other player has on the table. Thus, by looping through all other players' belief states, an estimate can be made for the total number of other 1/6s on the table. Combining this estimate with the POMDP's known information about its own hand, the POMDP can determine whether it should bid or challenge.

An estimate for the number of 1/6s a player on the table has can be generated by summing over all the number of 1/6s possibilities with the probability associated with that hand. The algorithm below shows the policy that we made for the POMDP model:

Algorithm 5 POMDP Policy

```
1:  function ACTIONPOMDP(state, belief)
2:      estimate ← Current 1/6s POMDP has
3:      for p in numPlayers do
4:          b ← belief[p]
5:          for i in possible 1/6s do
6:              estimate += b.vals[i] * b.probs[i]
7:          end for
8:      end for
9:      if estimate $<$ state.bid then
10:         return challenge
11:     else
12:         return bid
13:     end if
14: end function
```

The policy loops through all players in the game, and sums the 1/6s value multiplied by the associated belief up for all players. This estimate, combined with the known 1/6s the POMDP has, can then be compared with the current bid on the table to determine if the POMDP should challenge or increase the bid.

This policy is significantly better than just following the one third rule, assuming the belief estimates are accurate. Compared to the MDP, the POMDP has significantly more information about the state and other players, which should lead to better performance. However, the POMDP updates its belief by assuming other players on the table are following the one third rule, which may not be accurate.

C. Additional Strategies

Besides the above MDP and POMDP models developed, we also experimented with various modifications to the above models.

1) Bluffing: The first modification that was tested was adding a bluffing probability to the MDP models. Currently, the formulation for the MDP policy is just to always follow the one third rule, however, this doesn't replicate how many real life players play the game. Almost all players bluff, purposely make a statistically unlikely bid, at some point to try and gain an edge on the opponents by tricking them. We can easily implement this into the policy for MDPs by adding a percentage chance that if the action should be challenge, the MDP bids anyways. The modifications can be seen below in red:

Algorithm 6 MDP Policy - With Bluffing

```
1:  function ACTIONMDP(state)
2:      yourDice ← Current 1/6s MDP has
3:      totalDice ← Total number of dice in the game
4:      otherDice ← totalDice - yourDice
5:      estimate ← yourDice + $\frac{\text{otherDice}}{3}$
6:      bluffProb ← Hyperparameter value
7:      if estimate < state.bid then
8:          if rand() < bluffProb then
9:              return challenge
10:         else
11:             return challenge
12:         end if
13:     else
14:         return bid
15:     end if
16: end function
```

2) Particle Injection: The next modification that was tested was adding particle injections into the POMDP rejection particle filter. The rejection filter prunes out all dice rolls that don't align with the one third rule. This means that as the bidding increases, eventually the belief state for dice rolls containing 0 1/6s or only 1 1/6 becomes 0. While this may be accurate to following the one third rule, this doesn't account for players who differ from this policy or perhaps bluff. Also, as these states become completely pruned out of the belief state, the belief space contains less information and potentially hurts the performance of the POMDP. Thus, the following modification was implemented and tested in the belief update function:

Algorithm 7 POMDP Belief Update With Particle Injection

```
1:  function UPDATEBELIEF(state, bid, o)
2:      currNumDice ← player[o]'s number of dice
3:      totalDice ← Total number of dice in the game
4:      otherDice ← totalDice - currNumDice
5:      numParticles ← 100,000
6:      transProb ← Empty counting array
7:      injectionProb ← Hyperparameter value
8:      for numParticles do
9:          rollDice ← Randomly roll currNumDice
10:         num1/6s ← Count number of 1/6s in roll
11:         estimate ← num1/6s + $\frac{\text{otherDice}}{3}$          ▷ 1/3 rule
12:         if estimate < state.bid then          ▷ a = challenge
13:             if rand() < injectionProb then
14:                 transProb[num1/6s] += 1          ▷ Accept
15:             else
16:                 continue          ▷ Reject
17:             end if
18:         else
19:             transProb[num1/6s] += 1          ▷ Accept
20:         end if
21:     end for
22:     belief[o] += normalize(transProb) * belief[o]
23:     return normalize(belief)
24: end function
```

V. Results

A. MDP Model

The first objective of this project was to create a game environment with a MDP player utilizing the one third rule policy. We successfully achieved this goal, the results of the MDP model are shown below:

1) MDP vs MDP: To test the winning percentage of two MDPs playing against each other, we varied number of starting dice from 1 to 35 and calculated the winning percentage of the first MDP model, Figure 2 shows the results.

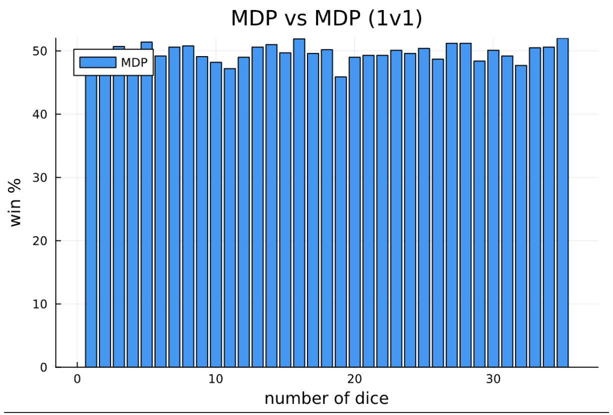


Fig. 2: MDP 1v1 win percentage vs number of game dice

When these players went face to face the game reaches an approximate equilibrium state where the win rate is 50 percent. This is the result that is expected considering both models are using the exact same strategy. This gives reassurance that our MDP model is performing the way that is expected.

2) MDP vs Manual Player: The MDP model was evaluated while playing a 1v1 against a human player. Figure 3 below shows the win rate average for 10 games with 7 dice.

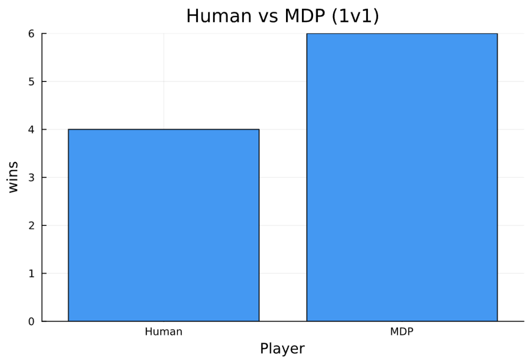


Fig. 3: Human against MDP with 7 starting dice

The results above show that the MDP is able to competitively compete against a human. These results show that an MDP model is able to perform quite well in Liar's Dice as statistically, it makes an accurate choice every time. Unfortunately, the sample size of this testing was limited. However, this small round of games suggests that an MDP is a fair baseline to compare the POMDP model with. It's performance is approximately on par with a standard human.

B. POMDP Model

The second objective of this project was to create a POMDP model that could beat a MDP model in a 1v1. We were able to successfully implement and simulate a POMDP model in our environment, our results for the performance of the POMDP are below.

1) POMDP vs MDP: To test the performance of our POMDP model vs our MDP model, we had both models start with the same number of dice and varied the starting dice from 1 to 35. We then simulated 1000 games for each number of starting dice and calculated the average winning percentage of the POMDP, Figure 4 shows the results.

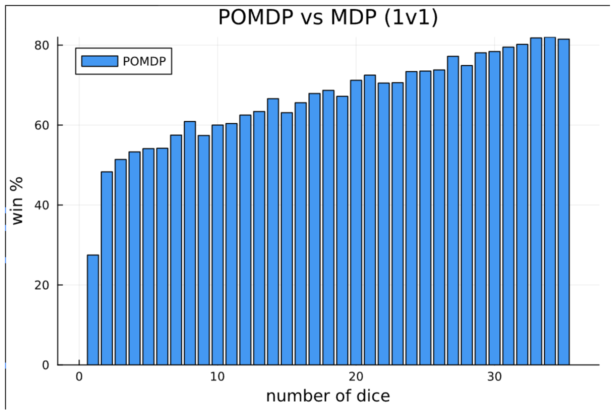


Fig. 4: POMDP wins against MDP vs number of starting dice

The POMDP performs at about a 50% win rate for a low number of dice, about 2-4 dice. However, as the number of dice increases, the POMDP begins to dominates the MDP, with a maximum win rate of 80% with 35 starting dice. This is because as the number of total dice increase, the POMDP has more opportunities to update its belief of the opponents hand, gaining more information. This updated belief gives the POMDP a massive advantage over the MDP as it can extract more information, allowing the POMDP to make a better decision on the action that should be taken.

2) POMDP vs Manual Player: The POMDP's performance against a human player was evaluated for 10 games with a starting hand of 7 dice each. Figure 5 below shows the results.

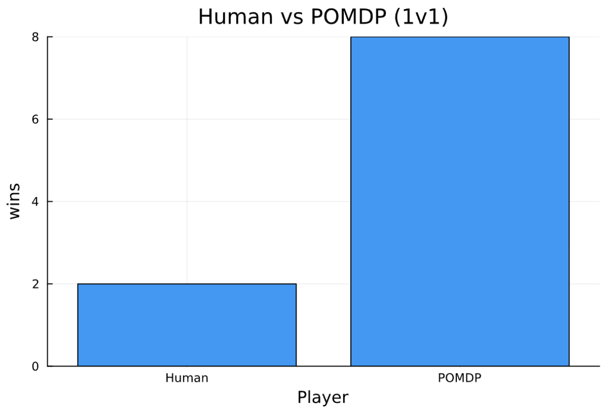


Fig. 5: Human against POMDP with 7 starting dice

The POMDP clearly outperforms a human with this small batch of experimental data. A quote from the user playing the POMDP was "it somehow knows exactly what dice I have, I don't get it". The user who played the POMDP generally followed the one third rule but did experiment with bluffing to try and trick the POMDP.

C. Scaling Up the Number of Players

The third objective of this project was to evaluate the performance of the POMDP model against multiple MDP models and manual players. We were able to successfully implement multiple players in our environment due to the formulation of the *play* function and assumptions reducing the state space size. The results are shown below:

1) POMDP vs MDP vs MDP: The POMDP's performance was evaluated against 2 MDP models following the "one third rule" strategy. The number of starting dice in a hand was varied from 1 to 35 dice. Each game setup was simulated 1000 times and POMDP's win percentage was calculated. The results from this benchmark test are shown in figure 6 below.

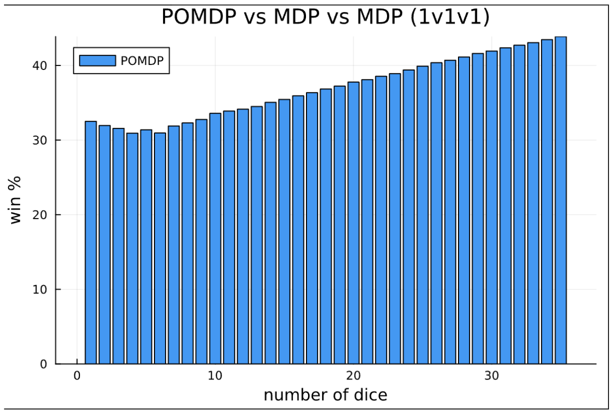


Fig. 6: POMDP wins vs game dice

The POMDP's was able to perform above an expected 33% win rate vs multiple MDP models. This shows that the strategy was more effective than probability alone, as used in the MDP models. Similarly to previous benchmarks, the POMDP became increasingly better as the number of dice in the starting hand increased.

Focusing on a game with 35 dice, it is clear that the POMDP's performance is much better than both MDPs. In this run the POMDP dominated the MDPs. The results are shown in Figure 7.

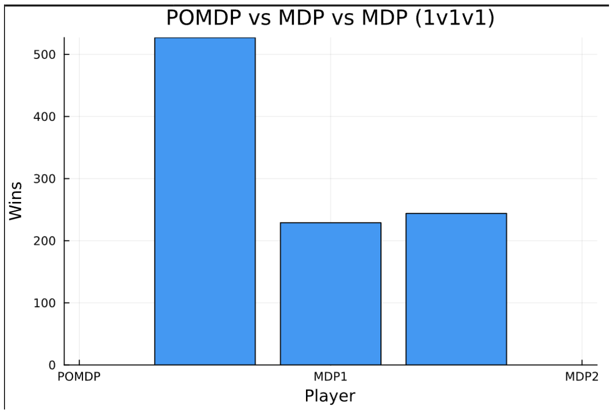


Fig. 7: Player vs number of wins for 35 dice game

The POMDP won 52.7 percent, MDP1 won 22.9 percent, and MDP2 won 24.4 percent of the games played. The reason as to why the POMDP preformed so well can be further analyzed by looking at the number of challenges from each model. This can help get a sense if certain agents over or under estimated the number of dice on the table. The number of challenges each player made is shown in Figure 8.

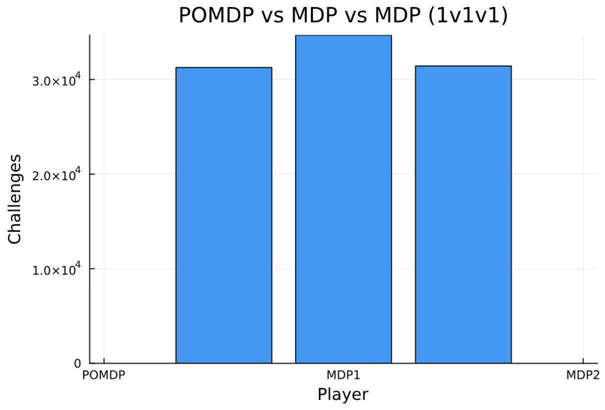


Fig. 8: Player vs number of challenges for a 35 dice game

MDP1 challenged significantly more than POMDP and MDP2. This shows that the POMDP consistently had beliefs that the hands of the other players exceeded the one third rule so the POMDP raised the bid. When it was MDP1's turn, it calculated that the POMDP's bid violated the one third rule so it challenged the POMDP. Overall, this challenge behavior of MDP1 was not successful as it had the lowest winning percentage of any of the three models.

2) POMDP vs MDP vs Manual Player: For the final experiment, a POMDP, MDP and Human went face to face in a three player game were each player has 7 dice to start. Figure 9 below shows the results of a 10 game match up.

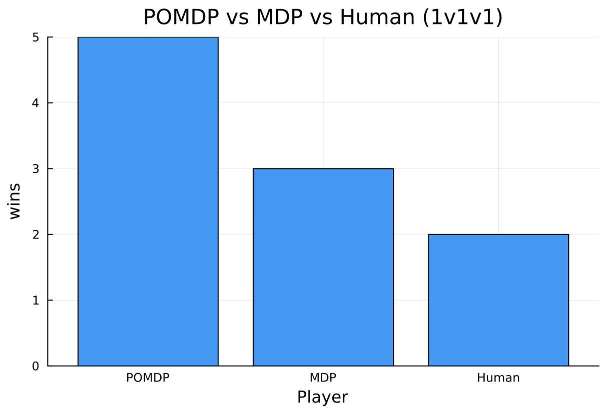


Fig. 9: POMDP vs MDP vs Human 21 with 21 starting dice

The results of this match up show just how good the POMDP is at estimating the other players hands. During this game play, oftentimes the human was eliminated from the game first and the POMDP would make easy work of the MDP. In games where the MDP was eliminated first the game was a little more balanced and a couple of rounds come down to each player having one dice.

D. Other Methods

1) Bluffing in MDP: To simulate the performance impact of MDP bluffing, we used a 1v1 environment between a POMDP and MDP, each starting with 15 dice. We then varied bluffing rate for the MDP and simulated each bluffing rate 1000 times. Below are the results:

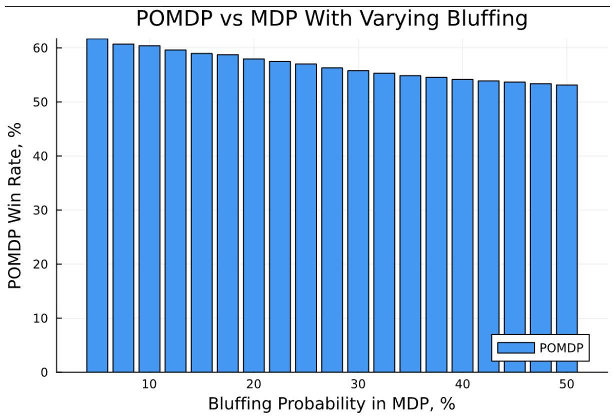


Fig. 10: POMDP vs MDP with varying bluffing, 15 dice

As evident in the results above, the POMDP model performs worse against the MDP model as the bluffing rate increases. This tells us that as the MDP model bluffs more, the POMDP model starts to believe that the MDP model must have really good dice, when in reality it doesn't. This causes the POMDP to perform slightly worse, although still better than 50% win rate.

These results make sense as the POMDP model was designed to play against players using the one third rule. As the MDP bluffs and differs from the one third rule, the POMDP belief update, which is rooted in the one third rule, becomes less accurate to the dice on the table. This highlights a flaw with the POMDP model, lying against it can be pretty effective at tricking to model into believing you have a good hand.

2) Particle Injection in POMDP: One method we experimented with to counter players lying or acting off-policy is to implement particle injection in the belief update of the POMDP. As discussed in the solution approach section above, particle injection helps reduce state pruning in our particle filter, allowing a higher variety of dice hands to be considered.

To test the performance of a POMDP with particle injection, we ran the same simulate as in the above section, a 1v1 between a POMDP and MDP, each start with 15 dice, with varying bluffing probability in the MDP. We wanted to test the performance of particle injection vs a MDP that lies/bluffs as the goal of particle injection is to counter opponents that play that way. Below are the results:

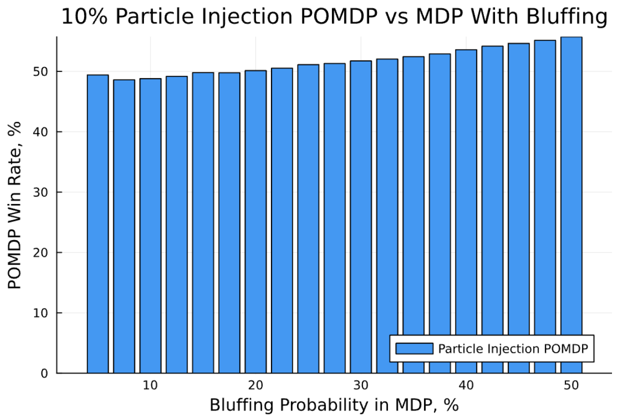


Fig. 11: 10% particle injection POMDP vs MDP with varying bluffing, 15 dice

Interestingly, the relationship between POMDP success and bluffing probability is completely opposite to the relationship seen prior in Fig 10 where the POMDP didn't have particle injection. In this simulation, the POMDP success rate increases as the MDP bluffs more. This tells us the particle injection does work to counteract the bluffing of another player. However, the overall success rate of the POMDP declined compared to not implementing particle injection. Before, the average success rate was around 55%, now, the average success rate is only 50%. This shows the particle injection may not be the ideal solution as it appears to decrease POMDP performance.

VI. Conclusion

In conclusion, we were able to successfully create a POMDP model that can play Liar's Dice and beat other models and human players at a higher than expect win rate. We successful met all three objectives of creating a simulation environment for Liar's Dice, implementing a MDP model using the one third rule, implementing a POMDP model that can outperform a MDP model, and expanding the game to have many players play against each other.

We believe the MDP model created represents a good baseline for how a standard human plays Liar's Dice, using raw statistics to estimate what the total number of dice on the table are. The performance of the MDP model vs a human had about a 60% win rate, although not many samples were taken. The win rate is probably around 50% if more samples were able to be taken. This is as expected and represents a very average AI model to play Liar's Dice.

The POMDP model created performed exceptionally well vs the MDP model and human players. In a 1v1 simulation vs the MDP model, the POMDP was able to win more than 60% of the time once the number of starting dice for each player reached 7+. As the number of starting dice increased, the POMDP eventually won over 80% of the time with 35 starting dice. This result makes sense as with a higher number of dice on the table, more observations are able to made throughout each turn, leading to increased information that the POMDP is able to obtain in it's belief update. In a sample set of 1v1s between the POMDP and a human player, the POMDP also performed exceptionally well. Although the sample set was only 10 games with 7 starting dice, the POMDP won 80% of the games vs a manual player.

We were also able to test the performance of the POMDP when opponents went off-policy relative to the one third rule. By implementing a chance of bluffing in the MDP model, we were able to simulate an off-policy model's performance vs the POMDP. Overall, as the MDP began bluffing more, the POMDP's performance decreased. This result makes sense as the POMDP model is assuming the opponent players are following the one third rule. However, the winning percentage of the POMDP was still greater than 50%, which is a successful model. To attempt to counteract opponents bluffing and possible particle depletion through the POMDP's particle filter, we also experimented with implementing particle injection. Our results for implementing particle injection into the POMDP model shows a net decrease in effectiveness of the model. The particle injection POMDP actually had a win rate of less than 50% in a few simulations, which we classify as a unsuccessful model.

Comparing our POMDP model with results generated from other solutions to Liar's Dice, such as Q-learning and Counterfactual Regret Minimization, our model is very competitive. The best model we could find which uses Counterfactual Regret Minimization had a 56.9% win rate vs human players, which is much lower than the win percentage our model is able to get, however, we had a small sample size. The best model we could find which uses Q-learning had a winning percentage of 65%, which is very successful and just a bit lower than the winning percentage our POMDP model was able to get.

While we did have great success in our POMDP model, the game of Liar's Dice was heavily simplified. By reducing the possible bids from any number to only 6s, we heavily reduced the state space and complexity of the game. However, because we are using particle sampling to update the belief of the POMDP, we believe the algorithms used could be expanded to a full game of Liar's Dice. The hardest part about expanding our models to a full game of Liar's Dice would not be getting an accurate belief update based on observations but it would be deciding which number to bid. For example should the model bid 4 2's or 4 3's? In the future, this problem could be considered and implemented into the MDP and POMDP model, allowing a simulation of the full game of Liar's Dice.

VII. Contributions and Release

A. Contributions

Chase Rupprecht	MDP 1/3 rule, Belief Update, Benchmarking
Nolan Stevenson	Code structure, POMDP, Particle Injection, Bluffing

B. Release

We would like to share our findings and release our write up to the public for future exploration.

References

[1] T. D. Ahle, "Liar's Dice by Self-Play," Towards Data Science, Jan. 16, 2022. https://towardsdatascience.com/lairs-dice-by-self-play-3bbed6addde0

[2] Sunberg, Zachary, et al. "POMDPs.jl: A Framework for Sequential Decision Making under Uncertainty." Journal of Machine Learning Research, vol. 18, no. 26, 2017, pp. 1–5, www.jmlr.org/papers/v18/16-300.html. Accessed 7 May 2024.

[3] T. W. Neller and S. C. Hnath, "Approximating Optimal Dudo Play with Fixed-Strategy Iteration Counterfactual Regret Minimization," *Advances in Computer Games*, pp. 170–183, Jan. 2012, doi: https://doi.org/10.1007/978-3-642-31866-515.

[4] D. Greaves, K. Tay, and T. Neller, "Winning Liar's Dice." Available: http://web.stanford.edu/class/archive/cs/cs221/cs221.1192/2018/

[5] Oehm, Daniel. "Q-Learning Example with Liar's Dice in R." Dan Oehm — Gradient Descending, 20 Dec. 2018, gradientdescending.com/q-learning-example-with-liars-dice-in-r/.