Exploring Algorithmic Approaches in Hanabi Gameplay

Jen Gordon, Sara McGee, Divyendu Shekhar, Jubal Bewick[1]

12/11/2024

[1]LLM agents assisted LaTeX formatting, grammar/spelling but did not assist in the formation and building of the proof or generation of the code.

Contents

1 Introduction **3**

1.1 Hanabi: A primer 3

2 Cheating/Perfect Knowledge **7**

2.1 Methodology 7

2.2 Deck Ceilings 7

2.3 Results 10

2.4 Clue Cost 12

2.5 Future Work 15

3 Mathematical Winability Analysis **16**

3.1 Problem Formulation 16

3.1.1 Min-Max Approach 16

3.1.2 Adaptation to Hanabi 16

3.2 Algorithm 17

3.2.1 Adaptation of the Min-Max algorithm 17

3.2.2 Implementation 17

3.3 Results 22

3.4 Limitations and Future work 23

3.4.1 Game state size and computability 23

3.4.2 Future Work 23

4 MWUM Learning Strategy **24**

4.1 Overview 24

4.1.1 Historical Context of MWUM 24

4.1.2 Cooperative Markov Decision Processes in Hanabi 25

4.1.3 Rationale for Using MWUM in Hanabi 25

4.2 Mathematical Foundations 26

4.3 Implementation Architecture 26

4.3.1 Weight Management 26

4.3.2 Action Selection 26
4.3.3 Weight Updates 27
4.3.4 Initial Attempts and Challenges 28
4.3.5 Enhanced Implementation Features 28
4.4 Experimental Results After Improvements 30
4.4.1 Performance with Enhanced Features 30
4.5 Conclusion 31

5 Githubs: 32

1 Introduction

This project aims to investigate and compare possible play strategies for the cooperative card game, Hanabi. The game presents unique challenges through its restrictions on game state information (cards in hand, cards in play, deck organization), player communication (allowable clues), and time of play (deck length). These constraints give rise to many different strategies which prioritize different actions. Our research intends to simplify these restrictions to compare strategies across various criteria.

Since communication is so limited in Hanabi, many players adopt conventions and assumptions to facilitate more predictable gameplay and expand the information able to be conveyed within these limits. We are investigating algorithmic approaches to play and hand management, focusing on deck organization and hand size as an information buffer.

1.1 Hanabi: A primer

As stated above, Hanabi is a cooperative card game in which the common goal is to play five colored suits of cards from 1 to 5 in the central play space (similar to Solitaire). In this report, we have represented these colors as red, orange, green, blue, and purple, for visibility's sake. Each suit contains three "1" cards, two "2"s, "3"s, and "4"s, and only one "5", as shown below.

1, 1, 1, 2, 2, 3, 3, 4, 4, 5

1, 1, 1, 2, 2, 3, 3, 4, 4, 5

1, 1, 1, 2, 2, 3, 3, 4, 4, 5

1, 1, 1, 2, 2, 3, 3, 4, 4, 5

1, 1, 1, 2, 2, 3, 3, 4, 4, 5

In the example below, Orpheus and Eurydice have taken a break from their epic quest to play a short game. We can see the opening setup below with both of their hands visible to us as a third party spectator.

Orpheus $\rightarrow$ 5, 2, 4, 4, 2

$Eurydice \rightarrow 5, 1, 2, 1, 5$

$misfires \rightarrow 0$

$clues \rightarrow 8$ $base \rightarrow 0, 0, 0, 0, 0$

$discard \rightarrow$

$deck \rightarrow 40$

During a game, a player cannot see their own cards, and can only see the cards of other players; therefore each player must clue cards in their co-collaborator's hands in order to successfully play cards into the center. Here we see Orpheus' view. He can only see Eurydice's hand, and nothing about his own.

$Orpheus \rightarrow *, *, *, *, *$ $Eurydice \rightarrow 5, 1, 2, 1, 5$

$misfires \rightarrow 0$

$clues \rightarrow 8$

$base \rightarrow 0, 0, 0, 0, 0$

$discard \rightarrow$

$deck \rightarrow 40$

Cards in hand can be clued either by color or number and each clue marks every card to which it applies, not just the desired target. The game starts with eight clues available; as clues are given, these tokens decrement. However, as cards are discarded, clues become available again, up to a full complement of eight. On each player's turn, they may play a card, discard (and earn a clue token), or give a clue (and spend a clue token).

In this example, Orpheus will choose to tell Eurydice about the 1s in her hand, using one of their clue tokens. The turn then passes to Eurydice.

$Orpheus \rightarrow 5, 2, 4, 4, 2$

$Eurydice \rightarrow *, 1, *, 1, *$

$misfires \rightarrow 0$

$clues \rightarrow 7$

$base \rightarrow 0, 0, 0, 0, 0$

$discard \rightarrow$

$deck \rightarrow 40$

Eurydice sees that she has two 1s in her hand that she knows are playable, and nothing useful in Orpheus's hand. Therefore, she plays one of her cards marked as a 1, and plays it. It is playable, so it is automatically placed in the correct color slot, and she draws a new card. It is now Orpheus's turn.

$Turn : Orpheus\ Orpheus \rightarrow *, *, *, *, *\ Eurydice \rightarrow 2, 5, 2, 1, 5$

$misfires \rightarrow 0$

$clues \rightarrow 7$

$base \rightarrow 0, 0, 0, 1, 0$

$discard \rightarrow$

$deck \rightarrow 39$

Orpheus now sees a number of green cards in Eurydice's hand. He clues the green cards in Eurydice's hand. $Turn : Eurydice\ Orpheus \rightarrow 5, 2, 4, 4, 2$

$Eurydice \rightarrow *, *, *, 1, *$

$misfires \rightarrow 0$

$clues \rightarrow 6$

$base \rightarrow 0, 0, 0, 1, 0$

$discard \rightarrow$

$deck \rightarrow 39$

Eurydice sees she definitely has a green 1 in her hand, and chooses to play it. The green on the base increments. $Turn : Orpheus\ Orpheus \rightarrow *, *, *, *, *\ Eurydice \rightarrow 5, 2, 5, 2, 5$

$misfires \rightarrow 0$

$clues \rightarrow 6$

$base \rightarrow 0, 0, 1, 1, 0$

$discard \rightarrow$

$deck \rightarrow 38$

At this point, Orpheus decides to discard, and earn a clue token back. He selects one of his cards, and puts it into the discard. The clues go up.

$Turn : Eurydice\ Orpheus \rightarrow 3, 5, 2, 4, 4,$

$Eurydice \rightarrow *, *, *, *, *$

$misfires \rightarrow 0$

$clues \rightarrow 7$

$base \rightarrow 0, 0, 1, 1, 0$

$discard \rightarrow 2$

$deck \rightarrow 37$

Eurydice decides to play her green card. Oh no! It's not yet playable. The misfire counter ticks up one, and the card is put in the discard. Misfires are analogous to "lives" or "strikes", and the games ends after three mistakes.

$Orpheus \rightarrow *, *, *, *, *\ Eurydice \rightarrow 2, 5, 2, 5, 2$

$misfires \rightarrow 1$

$clues \rightarrow 7$

$base \rightarrow 0, 0, 1, 1, 0$

$discard \rightarrow 2, 5$

$deck \rightarrow 36$

Game end is triggered when all the cards in the deck have been drawn and each player has one final turn to complete their play. In the example below Eurydice is on her final turn and although she has two 5s marked in her hand, she will only be able to play one of them before the game ends.

$Orpheus \rightarrow 3, 1, 2, 4,$

$Eurydice \rightarrow *, *, 5, 5, *$

$misfires \rightarrow 1$

$clues \rightarrow 3$

$base \rightarrow 4, 4, 5, 5, 5$

$deck \rightarrow 0$

In the experiments below, we have each simplified some of these restrictions to isolate specific strategy variables. However, the full game strategy consists of trade-offs between hidden information, restricted communication, and the limit of the deck.

2 Cheating/Perfect Knowledge

2.1 Methodology

2.2 Deck Ceilings

There are a nearly unlimited number of conventions in Hanabi. One could perhaps, specify that cluing "Blue" meant to play the third card, followed by discarding the fourth. It is an immediately interesting question to ask which of them is the best.

In order to be able to judge a clue strategy, it is important to be able to answer on average what the maximum score is in Hanabi. It is readily apparent that not every deck of Hanabi is winnable with a perfect score.
The goal of my first exploration was to try to establish some sort of score ceiling for two players. Suppose then that both players through some cluing strategy always have perfect knowledge of each other's hands. What would be the optimal strategy?

Let us explore some ways that such players could still fail to achieve a perfect score. First, we could be forced to discard an essential card. We call a card essential if it is the only card

of that type available to be played for the rest of the game. Fives, because of the makeup of the deck, are always essential.

Imagine a deck conversationally constructed like so:
$bottom \rightarrow 1, 1, 1, 1, 1, 1, 1, 1, 1, ...5, 5$

The two players have a combined hand space of 10 cards. Even with perfect knowledge, they would be forced to discard an essential card. They could save all essential fours and fives but then would have no hands space left over. To progress, they would be forced to discard an essential card, thus reducing their max score.

There are more nuanced versions of this problem as well Consider the following game state:

$base \rightarrow 1, 0, 1, 5, 1$
$p1 \rightarrow 3, 4, 5, 3, 4$
$p2 \rightarrow 5, 5, 3, 3, 4,$

$discard \rightarrow 2, 2, 3, 4, 2, 3, 4$

In this situation, both players hands have become filled up with essential cards, but not necessarily by necessity. In the discard were the 2 and 2. If they were kept then the $4, 5, 3$,3 would be playable.

The two players could also run out of time to play all essential cards. Consider the following situation:
$p1 \rightarrow 1, 4, 4, 2, 3$
$p2 \rightarrow 1, 3, 5, 4, 3$

$base \rightarrow 1, 3, 5, 3, 1$
$deck\ bottom :\rightarrow 4, 5$

In this situation, the limiting factor isn't hand size, but time. Whoever draws the blue four will trigger the end of the game in 2 moves. While they can play the 4, no one will be able to play 5, as the game will end.

To avoid these pitfalls as much as possible, my strategy does actions in the following priority:

1. If there is a playable card in either player's hand, either play the card (if in the current hand) use a clue to pass the turn to the other player to play it.
2. If there is a card either that we have two of in hand, or a card lower in the color than what has been played (e.g. a 3 when the 4 as been played), we either discard it, or pass the turn for the other player to discard.
3. From there, we look at non-essential cards, and determine which of them is furthest away from being played, and either discard that one, or pass the turn for our partner to discard it.
   (a) This requires some more explanation. Consider the following situation. We need to choose between discarding the 3 and the 4. Now, we have an equal chance at drawing any card left in the deck $p = 1/len(deck)$. But for any two cards that probability goes down to $p = 1/len(deck) * 1/len(deck) - 1$
   . If we try to save the 4, we'll only need to draw one card to play it, whereas if we try to save the 3, we'll need to draw two cards to play it. Therefore, in order to maximize the likelyhood of our hand filling up with essential cards, we discard the 3, even though it has a larger value than the 4.
   $p1 \rightarrow 2, 3, 4, 3, 4$
   $p2 \rightarrow 5, 5, 4, 4, 5$

   $base \rightarrow 2, 2, 0, 0, 0$ $discard \rightarrow 2, 3, 4, 4, 4$
4. If there are only essential cards in both players hands, we discard the highest, as that

has the least effect on our max score.

2.3 Results

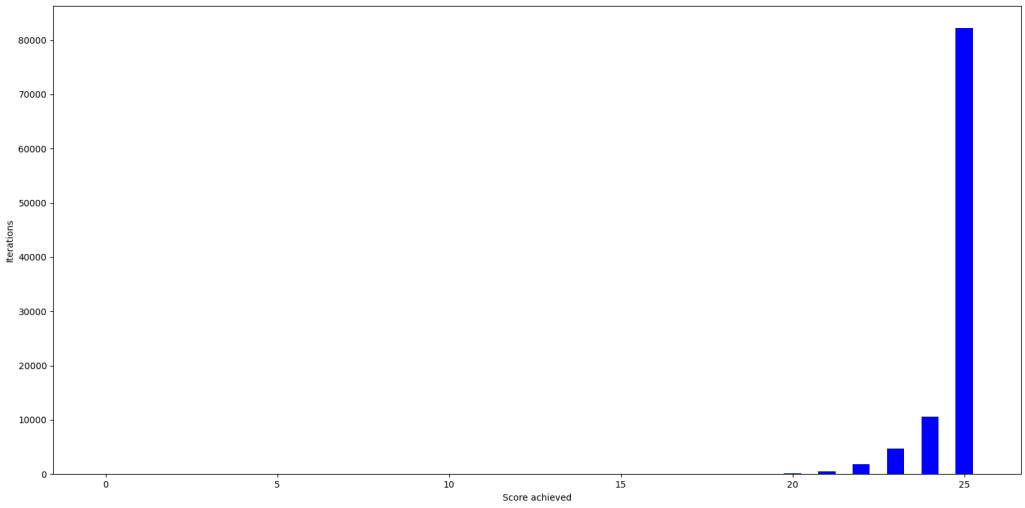


Figure 1: Winnability

Our results are quite good over 10^5 iterations, The average score of this version of the game is 24.8744 While this is not a perfect score, it very nearly is.

This result puts a lot of weight on clue strategies. Essentially, if the maximum average score in Hanabi.

We might think that this is an unreasonable amount of information to expect any strategy to give. Suppose we replaced the third step in the above algorithm with simply discarding the oldest non-essential cards. Even this strategy performers exceptionally well, with an average score of 24.74596.

Further, optimizing the discard has relatively low gains compared to saving essential cards and marking playable cards.

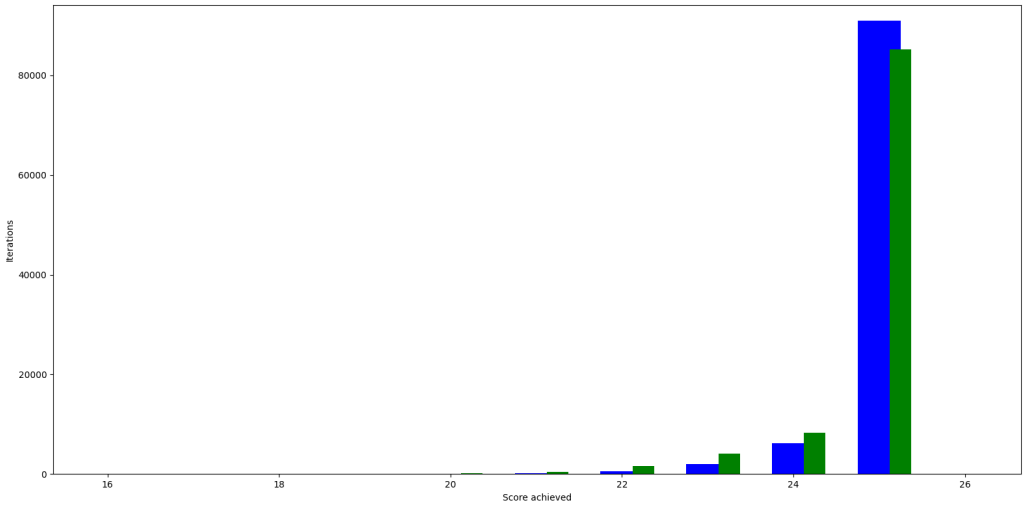


Figure 2: Discard Optimization

Given these near-perfect results, we became curious about the theoretical maximum including cluing constraints. We developed a derivative strategy that requires each card to be clued by at least one attribute before it can be played. Each game starts with eight clue tokens available and forces a discard when each of these clues have been used. As usual, each discard earns back one clue, and players cannot earn more than eight clue tokens, but we did allow additional discarding even with all eight clues available. For the benchmark maximum, the cluing strategy prioritizes the greatest number of cards included in a single clue. Each clue given was required to include a playable card, but, since we're still interested in theoretical maximums, we retained perfect knowledge, assuming players could determine by convention which cards might be playable when several were clued.

Therefore, on each player's turn, first consider playing any playable marked cards in hand. If none are playable, and clues are still available, then iterate over the other player's hand, tallying up the cards included in each clue set. Then find the maximum clue set which includes a playable card and give that clue. If there are no playable cards to be clued, or no clues available to be given, discard. This algorithm takes a somewhat greedy approach, only

considering currently playable cards, but also maximizing cards clued per clue, in the hopes that as cards become playable in the future, they will have already been clued.

We were surprised to find that even with cluing as a limiting factor, the average score was still around 24.5. The games took about twice as long, but ultimately clue constraints didn't significantly hamper play in a perfect knowledge game.

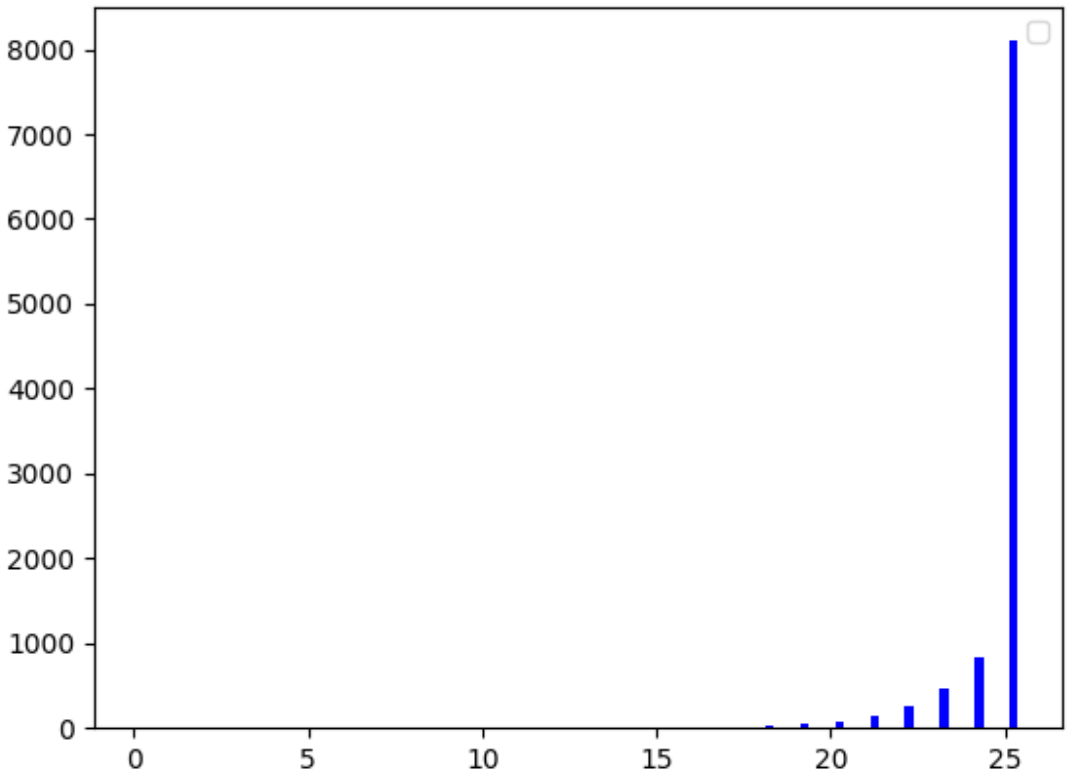


Figure 3: Clue Efficiency Maximum

2.4 Clue Cost

As has been briefly discussed, there are a limitless number of conventions you could follow in Hanabi.

We could begin to compare clue strategies with the idea of clue efficiency. Consider the following situation: $p1 \rightarrow 4, 4, 4, 4, 4$
$p2 \rightarrow 1, 1, 3, 1, 1$

$base \rightarrow 0, 0, 0, 0, 0$

Player 1 has the chance to give Player 2 an incredible clue. If Player 1 clues all the '1's in Player 2's hand, that will be all the information player 2 needs to play a substantial number of cards.

How good is this clue? One way we could begin to quantify this clue was to denote the the number of clues to communicate / number of cards communicated as playable. In this case 1 clue marked 4 cards for a play cost of .25.

In order to get a sense of how efficient our clues need to be, we might add an arbitrary cost to playing a card, while still giving perfect knowledge to both players. It takes 2 clues to completely reveal a card. A reasonable range of clue costs to explore would be between 0 and 2.

We can modify the algorithm used in the last section with relatively few changes (The only one being logic to not clue the other player if that would result in the other player not having enough clues to play)
On doing that, we achieve the following result.

These results really reaffirm the idea that the main thing holding back scores in Hanabi are the number of clues. Total scores go down linearly as the cost of clues goes up. In other words, the more clue efficient a strategy can be, the greater the expected average score.

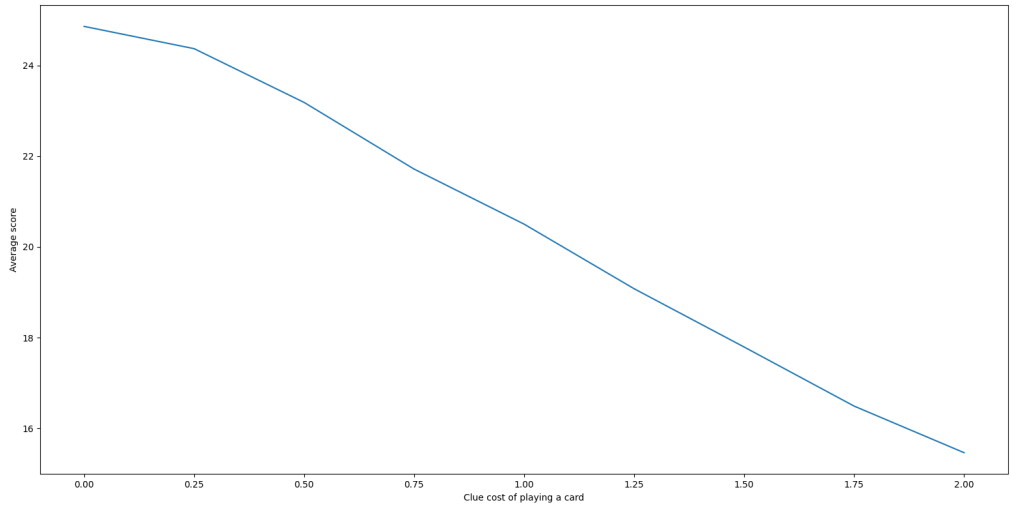


Figure 4: Enter Caption

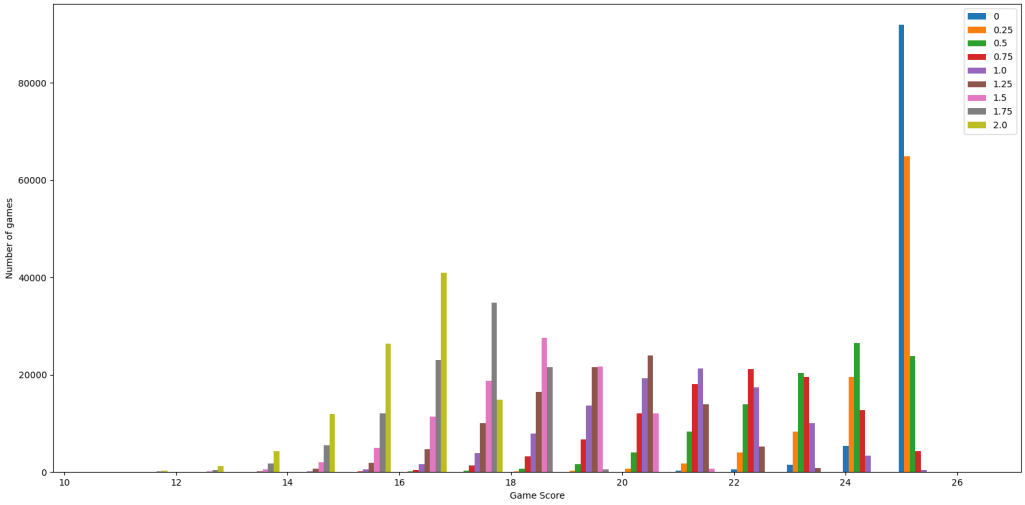


Figure 5: Clue costs

To this end, we began to consider building out cluing conventions by prioritizing either color or number clues and began to wonder how the two strategies might compare. We set up a similar approach as above, but rather than maximizing, simply clued playable cards by either color or number. Surprisingly, both approaches perform pretty similarly, each produc-

ing average scores around 23.5. However, from the graphs, it appears that cluing by color achieves a perfect score more often while cluing by number scores across a wider distribution.

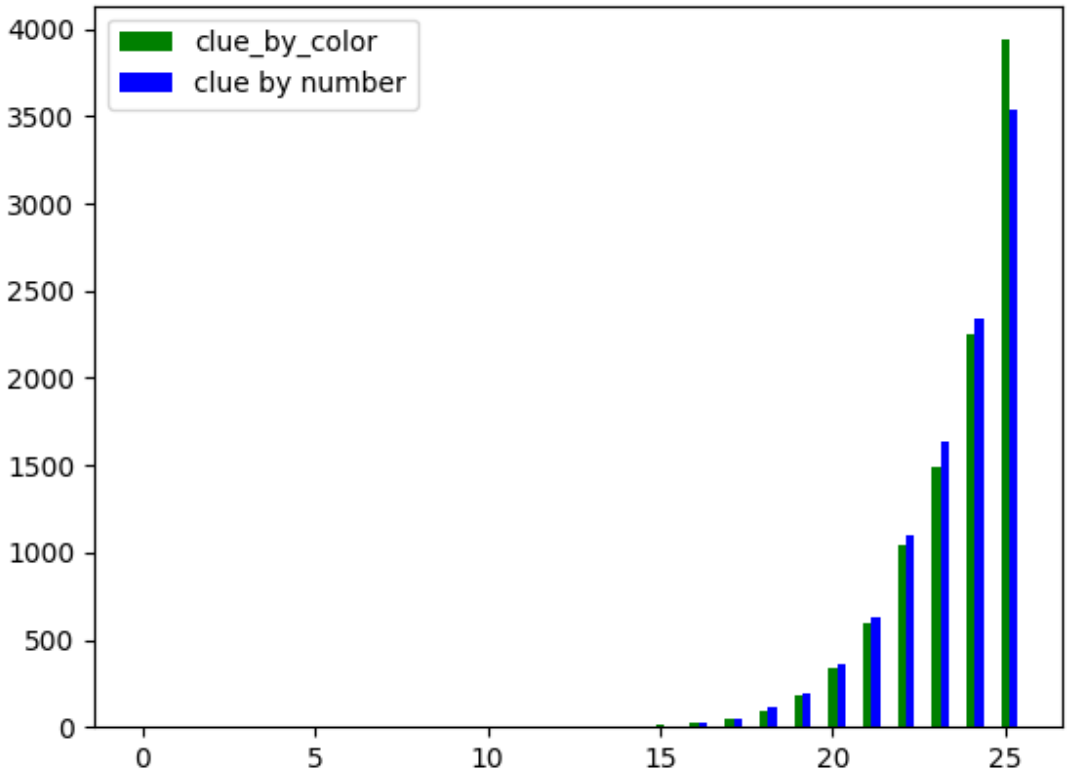


Figure 6: Color vs. Number Cluing Comparison

2.5 Future Work

This work lays an important foundation for analyzing strategies in Hanabi. It gives us a real ceiling of what we can expect any clue strategy to attain, and provides some tools to measure how effective a particular strategy is.

There are some flaws yet in the second metric. Passing the turn back and forth always

takes a clue. Especially in the less efficient strategies, it might be worth taking a less efficient discard rather than using a clue token to pass the turn. Finding where exactly this balance in would require extensive further testing and models.
There remains much work to do analyzing whether there exists such a strategy that can achieve an average clue efficiency of .25.

3 Mathematical Winability Analysis

3.1 Problem Formulation

In Hanabi, the randomness of deck shuffles introduces significant uncertainty to game-play outcomes. Some shuffles may be unwinnable, and even with perfect play may lead to a not-perfect score.

3.1.1 Min-Max Approach

To evaluate whether a Hanabi deck shuffle is winnable, we adapt the Min-Max tree, a well-established algorithm in game theory. Traditionally, the Min-Max tree is used in adversarial games to model decisions made by two players with opposing goals. Each node in the tree represents a game state, while edges correspond to possible moves. The algorithm alternates between minimizing and maximizing strategies:
Maximizing Player: Attempts to choose moves that maximize their score or advantage.
Minimizing Player: Assumes the opponent will respond with moves that minimize the first player's advantage.
This approach ensures robust decision-making by accounting for the "worst-case" opponent behavior.

3.1.2 Adaptation to Hanabi

Unlike adversarial games, Hanabi is a cooperative game where all players work towards achieving the maximum possible score. Therefore, the traditional Min-Max can't be applied directly. Instead, we can modify the algorithm by maximizing the score at every step, re-

flecting the shared goal of the players. This eliminates the need for alternating minimization layers and aligns the algorithm with the nature of cooperative game-play.

3.2 Algorithm

3.2.1 Adaptation of the Min-Max algorithm

The Min-Max tree traditionally evaluates all potential game states for adversarial decision-making. For our specific problem, we modified it to maximize the score across possible game outcomes given a shuffled deck.

- State Representation
  - Each node in the tree represents a specific game state: the current hand, the state of the play area, the discard pile, and the number of clue.
  - Transition edges represent valid player actions (e.g., hinting, discarding, or playing a card).
- Score Maximization
  - Since Hanabi is a cooperative game, instead of minimizing and maximizing at alternate depths of the tree, we maximize the score regardless of the turn.

3.2.2 Implementation

Each game state stores the following knowledge:

- Board - Current board which stores all the played card for all the colors.
- PlayersHands - All the cards in each players' hand.
- RemainingCards - Remaining card in the deck the players can draw from.
- DiscardedPile - Pile of discarded cards.
- Hints - Hints allowed.
- CurrentPlayer - Player whose turn it is.

- Score - current score of the game state.
- MaxPossibleScore - Maximum possible score achievable = Total colors in the game * maximum value of card in each color.

Each Card has:

- Color - Color of the card.
- Number - Number of the card.

Each Move has:

- Type - hint, play, or discard.
- Card - If play or discard, which card and by which player.

The Evaluate procedure is the main entry point to assess the winability of a game state by leveraging a modified maximize-tree algorithm.

- We initialize bestScore to 0. This variable stores the highest score during the evaluation of all possible moves.
- The LegalMoves() gets all possible legal moves from the current game state. If no moves are available (indicating that the game has reached a terminal state), games current score is returned.
- For each legal move: - We push the move onto the current game state to get a new game state. - The Evaluation function is called recursively to calculate the potential score for the new game state.
- If the calculated score for a move is equal to or exceeds the maximum possible score of the game (MaxPossibleScore), the algorithm immediately returns this score. This pruning ensures that the algorithm terminates as soon as it identifies a move that guarantees a win or the highest achievable score.
- If the calculated score is greater than the current bestScore, it updates bestScore to this value.

- After evaluating all legal moves, the function returns bestScore, representing the highest achievable score for the current game state.

Algorithm 1 Evaluate winability

```
1: procedure EVALUATE(game)
2:     bestScore ← 0
3:     moves ← game.LegalMoves()
4:     if len(moves) = 0 then
5:         return game.score
6:     end if
7:     for all move ∈ moves do
8:         newGame ← game.pushMove(move)
9:         score ← Evaluate(newGame)
10:        if score ≥ game.MaxPossibleScore then
11:            return score
12:        end if
13:        if score > bestScore then
14:            bestScore ← score
15:        end if
16:    end for
17:    return bestScore
18: end procedure
```

PushMove procedure plays the move and then returns a new updated game state resulting from the played move.

- We first check if the move is a hint, if yes, we decrement the hint to get the new game state.
- If move is not a hint, it must be a play or a discard. Regardless, we remove it from the player's hand.
- Now, if the move is play, we increment the game score by 1.

- else if the move is discard, we move card to discard pile and increment game hint by 1.
- In both the case if the move is discard or play, if there are cards in the remaining deck, we add it to player's hand.
- Finally we change the current player and return the game.

Algorithm 2 PushMove

```
1: function PushMove(game, move)
2:     if move is Hint then
3:         game.Hints ← game.Hints − 1
4:     else
5:         Remove move Card from current player's hand
6:         if move is Play then
7:             game.Score ← game.Score + 1
8:         else if move is Discard then
9:             Add move Card to game.DiscardedPile
10:            game.Hints ← game.Hints + 1
11:        end if
12:        if |game.RemainingCards| > 0 then
13:            Pop top card from RemainingCards and add to current player's hand.
14:        end if
15:    end if
16:    game.changePlayer()
17:    return game
18: end function
```

LegalMoves procedure generate legal moves the player can play. We don't generate moves for misfires as the algorithm has perfect knowledge and we are trying to maximize the score.

- We declare 2 lists - playableMoves and discardMoves.
- if hints remaining for current game state is ¿ 0, we add it to discardMoves. we do so as hinting is used to simply change players and doesn't actually have any other purpose.

- We go over every card in player's hand and if a card can be played on the board, we add it to playableMoves.
- else we add the card to discardMoves.
- Finally, if we have playable moves, we return legal moves as playableMoves. Since we can be greedy in Hanabi if we have perfect knowledge.
- Otherwise is we don't have any playable moves, we return discardMoves.

Algorithm 3 Generate all legal moves we can play

```
1: function LegalMoves(game)
2:     playMoves ← [ ]
3:     discardMoves ← [ ]
4:     if game.Hints > 0 then
5:         discardMoves ← Hint move
6:     end if
7:     for all card ∈ current player's hand do
8:         if card is playable without a misfire then
9:             Add card to playMoves
10:        else
11:            Add card to discardMoves
12:        end if
13:    end for
14:    if |playMoves| > 0 then
15:        return playMoves
16:    end if
17:    return discardMoves
18: end function
```

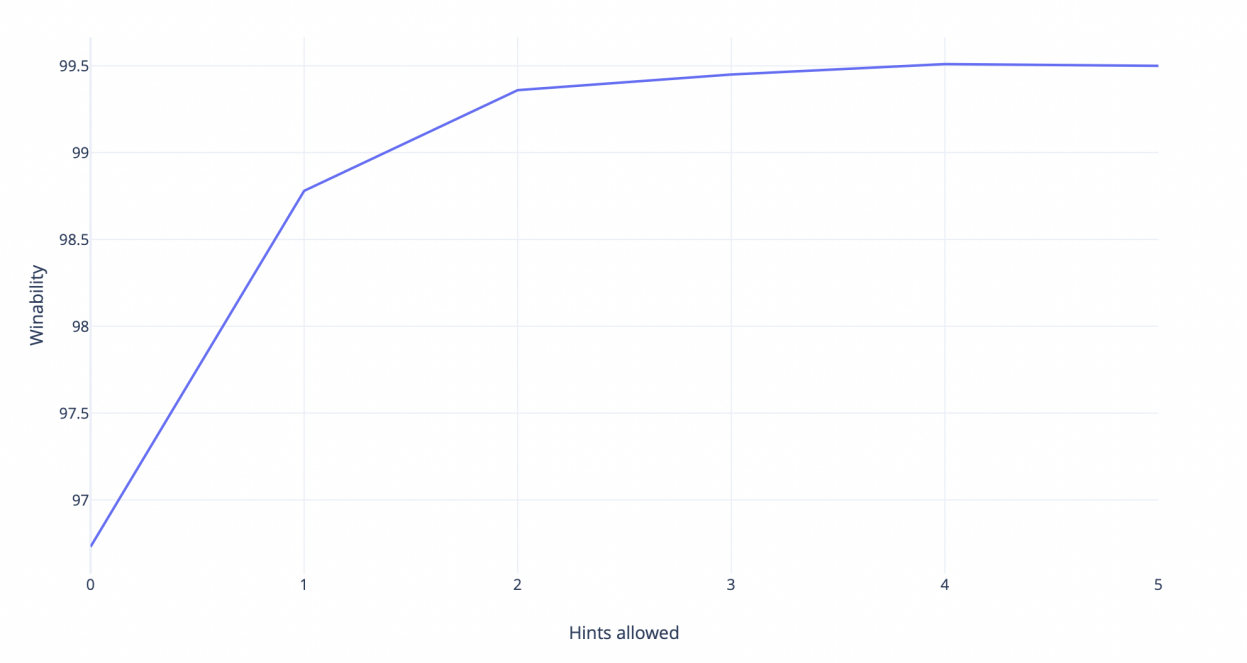
3.3 Results

To validate the effectiveness of the modified Min-Max tree approach, we tested its performance under controlled conditions. The parameters were as follows:

- **Number of players:** 2
- **Number of colors:** 3
- **Each player's hand size:** 3 cards
- **Deck composition:** Each color has 2 1s, 2 2s, and 1 3s, therefore total number of cards = 15

The results are summarized in the table below, showing the frequency of achieving a perfect score and near-perfect scores for various allowed total hints over 100,000 runs.

Hints	0	1	2	3	4	5
Perfect Score	96729	98785	99360	99454	99506	99496
Perfect Score - 1	3251	1190	618	516	470	477
Perfect Score - 2	20	25	22	30	24	27
Perfect Score %	96.73%	98.78%	99.36%	99.45%	99.51%	99.50%



As the number of allowed hints increases, the percentage of perfect scores also increases. With 0 allowed hints, 96.73% of the decks are winnable with perfect play, while with 3 hints allowed, the winability improves to 99.45%. However, we see there is a plateauing of winability as the hints increase above a certain threshold, 3 in our current scenario.

3.4 Limitations and Future work

3.4.1 Game state size and computability

Unlike traditional Min-Max tree/ Alpha-Beta pruning algorithm, We don't have a clear-cut way of pruning useless branches. This leads to an exponentially large number of game states that needs traversing. This lead to us needing to run this algorithm on a reduced set of criteria instead of the standard ruleset and criteria for a Hanabi game.

3.4.2 Future Work

We could add further pruning by simplifying the result to Winnable or not-winnable. - If all cards of same color and number are discarded, we say the game is unwinnable. If we reach

perfect score, the game is winnable. This could lead to a significant improvement to the runtime as we will prune significant amount of branches which we are currently searching to get not-perfect scores.
We could also convert the current structure of game to use bit-boards to store data, which might result in very efficient LegalMoves and PushMoves functions.
Our algorithm is generic and can be extended to multiple players and variation of cards with more colors and numbers, with only hurdle being the runtime.

4 MWUM Learning Strategy

4.1 Overview

Building on Arora et al.'s foundational work [1], I implemented the Multiplicative Weights Update Method (MWUM) to develop and understand agent-based strategies in Hanabi. MWUM is a seminal work in game theory, optimization and machine learning for its capacity to minimize regret over time and adjust strategies adaptively as more information is gathered. It should be noted it is not the most optimum or advanced algorithm anymore but it pivotal in the role it has played. Its well-characterized theoretical properties make it a natural candidate for tackling a complex, partially observable and cooperative environment like Hanabi without having to build a neural network and introduce transformers.

4.1.1 Historical Context of MWUM

The conceptual roots of MWUM stretch back to the 1950s. G.W. Brown's fictitious play [4] introduced multiplicative updating for iterative strategy refinement. J. Hannan's consistent estimator [5] integrated the idea of regret minimization into repeated decision-making scenarios; a principle now central to machine learning algorithms that learn from ongoing feedback.

By the 1980s, Littlestone and Warmuth's Weighted Majority Algorithm [6] influenced the field by weighting multiple experts' predictions and laying groundwork for adaptive data-

driven strategies. Freund and Schapire's AdaBoost [7] in the 1990s further showcased multiplicative updates – this time combining weak learners into a strong ensemble. Later Arora, Hazan, and Kale [1] formalized MWUM as a meta-algorithm revealing its broad applicability from linear programming approximations to mastering zero-sum games.

4.1.2 Cooperative Markov Decision Processes in Hanabi

My goal was to try and adopt Littman's Markov game framework [3] to model Hanabi as a cooperative Markov decision process (CoMDP). Two agents (players) share a single objective: to maximize the game's final score. Hanabi's hidden-information setup, restricted clues and collective reward signal align well with CoMDP structures.

- **Multiple Agents:** Both players impact the shared state through their actions.
- **Shared Reward:** A unified objective (maximize the final score) encourages synergy.
- **Joint State Space:** State captures both players' hands (hidden from themselves and knowledge of the others hand), the discard pile, ongoing plays, misfires and clue tokens.

4.1.3 Rationale for Using MWUM in Hanabi

MWUM's adaptability and regret minimization are key, along with it being a seminal algorithm that I wanted to experience working with myself:

- **Adaptability:** Hanabi's hidden information and sparse feedback demand a strategy that improves incrementally as outcomes are observed.
- **Regret Minimization:** To ensure that over many plays the chosen strategy approaches the performance of the best fixed action in hindsight.
- **Theoretical Guarantees:** Provides a conceptual framework for understanding performance bounds even if it remains challenging to bridge the gap between theory and practical implementation.

4.2 Mathematical Foundations

At each time step the Multiplicative Weights Update Method (MWUM) updates a probability distribution over actions based on their historical performance following these steps:

1. Maintain weights w_t^i for each action i.
2. Compute $p_t^i = \frac{w_t^i}{\sum_j w_t^j}$ to form a probability distribution.
3. Select an action according to p_t.
4. Observe a reward/loss r_t^i.
5. Update weights $w_{t+1}^i = w_t^i \exp(\eta r_t^i)$.

This mechanism goal is to attempt to ensure that repeatedly successful actions grow more influential over time while maintaining a capacity for exploration.

4.3 Implementation Architecture

4.3.1 Weight Management

For each encountered state I maintain a set of weights for all valid actions. When a new state is observed for the first time all valid actions are initialized with equal weights with the goal of fostering exploration by ensuring no single action is prematurely favored.

Listing 1: Weight Initialization Function

```
def get_action_weights(self, state_hash, valid_actions):
    if state_hash not in self.weights:
        self.weights[state_hash] = {action: 1.0 for action in valid_actions}
    return self.weights[state_hash]
```

4.3.2 Action Selection

Actions are selected according to a probability distribution derived from their weights. Without this stochastic element the agent would consistently exploit the action with the highest weight and would be missing opportunities to explore potentially better alternatives.

Listing 2: Action Selection Function

```
def choose_action(self, weights):
    actions = list(weights.keys())
    probs = np.array(list(weights.values()))
    probs = probs / probs.sum()
    return np.random.choice(actions, p=probs)
```

Why This Matters:

- **Balancing Exploration and Exploitation:** A purely greedy approach might lock onto a single action prematurely with theoretically would lead to stalling further improvement. By selecting actions probabilistically the agent continues to sample a variety of options.
- **Progressive Convergence:** Over time higher-performing actions accumulate greater weights and thus higher probabilities of selection. However the probabilistic mechanism ensures that other actions still have occasional chances by preventing the agent from getting stuck too early.

4.3.3 Weight Updates

At the end of each episode (one complete run of the game) the weights are updated using the final score as a reward signal:

Listing 3: Weight Update Function

```
def update_weights(self, reward):
    for state_hash, action in self.game_history:
        self.weights[state_hash][action] *= np.exp(self.learning_rate * reward)
    self.game_history = []
```

What is an Episode? An episode begins when a game starts and ends upon a terminal condition (e.g. all plays made or running out of misfires). The final score then guides how the weights are adjusted to learn from the entire sequence of actions taken.

4.3.4 Initial Attempts and Challenges

In early experiments, I ran 100,000 episodes, testing different learning rates ($\eta = 0.1$ and $\eta = 0.5$). The resulting performance consistently hovered around 1.2–1.3 points out of 25. This stability suggested that simple parameter tuning was insufficient to overcome Hanabi's complexity and the sparse, delayed nature of its reward.

4.3.5 Enhanced Implementation Features

To address these shortcomings I introduced multiple enhancements:

Improved State Representation I incorporated a bucketing strategy and distilled the state into select features that capture the essential dynamics of the game. This approach I hoped would help the agent generalize more effectively across various situations.:

Listing 4: Enhanced State Representation

```
def hash_game_state(self, my_hand, other_hand, discard, play_base, misfires, clue_tokens)
    :
    playable_count = sum(1 for card in other_hand if card.value > play_base[card.color])
    hand_values = tuple(sorted(card.value for card in my_hand))
    remaining_plays = sum(1 for base in play_base for val in range(base + 1, 6))
    discard_risk = sum(1 for val, count in discard.items() if count >= 2 and (val % 5) !=
         0)

    state = (
        hand_values,
        min(playable_count, 3),
        remaining_plays,
        min(discard_risk, 3),
        min(2, misfires),
        min(clue_tokens, 4)
    )
    return str(hash(state))
```

This abstraction aimed to reduce the effective dimensionality of the state space by making

it "easier" for the agent to identify patterns.

Advanced Reward Shaping I modified the reward structure to guide the agent's learning by providing incremental feedback, rather than relying solely on the final score, thereby highlighting ongoing improvements and reinforcing steady progress.:

Listing 5: Enhanced Reward Calculation

```
def calculate_reward(score, prev_scores):
    base_reward = score / 25.0
    recent_avg = np.mean(prev_scores[-100:]) if prev_scores else 0
    improvement = score - recent_avg if prev_scores else 0

    if score < 5:
        reward = -1.0
    elif score < 10:
        reward = base_reward * 0.5
    elif score < 15:
        reward = base_reward * 1.5 + (0.1 * (improvement > 0))
    else:
        reward = base_reward * 3.0 + (0.2 * (improvement > 0))
    return reward
```

This was intended to guide the agent more effectively by offering intermediate signals of progress.

Adaptive Learning Rate I also introduced an adaptive learning rate mechanism so that states visited frequently would have more cautious updates:

Listing 6: Adaptive Learning Rate

```
def update_weights(self, reward):
    self.epsilon = max(self.min_epsilon, self.epsilon * self.epsilon_decay)

    for state_hash, action in self.game_history:
        visit_count = self.state_visits[state_hash]
        adaptive_lr = self.learning_rate / np.sqrt(max(1, visit_count))
```

```
7
8           multiplier = 1.0 + adaptive_lr * reward
9           if reward < 0:
10              multiplier = 1.0 / (1.0 - adaptive_lr * reward)
11
12          self.weights[state_hash][action] *= multiplier
```

The aim was to stabilize updates over time and prevent overly aggressive changes in well-explored states.

4.4 Experimental Results After Improvements

4.4.1 Performance with Enhanced Features

With the improved state representation, advanced reward shaping, and adaptive learning rate I trained the model for another 100,000 episodes. It should be noted it reach local minima or maxima by 11000 episodes and ended. Here is the observation:

- Initial performance around 2.00 at episode 0 (reflecting starting conditions).
- Stabilized performance around 1.26–1.28 by episode 1000.
- A final stable performance near 1.29 after 5000 episodes.

While this slightly surpassed the previous 1.2–1.3 range it remained far from my goal exploring high point value strategies without direction.

Learning Dynamics In the early stages of training the agent's high exploration rate (epsilon) prompted it to try a wide variety of actions. While this broad search sampled numerous potential strategies none translated into consistently better outcomes—average scores struggled to surpass even 1.3 out of 25 points. As epsilon decreased over time I believe the agent began to rely more heavily on whatever limited successes it had found even though those successes were marginal. This shift may have led to premature convergence on policies that offered slight short-term advantages over random play but did not pave the way to genuine improvement. It should also be noted the scarcity of data on which it could

train itself along with local optimums being present most likely weighed heavy on the agents ability to optimize.

By around 1,000 episodes early gains had plateaued and the simulation after updated ended by 11000 episodes due to plateauing. Parameter refinements like enhanced reward shaping or adaptive learning rates—tools intended to refine the learning process—did not spark further progress. The agent appeared trapped in a local optimum, as evidenced by the very early plateauing, reinforcing patterns that exploited narrow context-specific cues but failed to generalize into more robust strategies. This stagnation can be attributed in large part to the complexity and partial observability of Hanabi. The scarcity and delay of meaningful feedback meant that incremental improvements were often too subtle or infrequent to steer the agent away from these low-value stable states.

The persistence of such low-level behavior, even when guided by theoretically sound algorithms like MWUM, underscores the challenges of escaping local optima in domains characterized by uncertainty, limited communication and delayed or sparse rewards. Incremental parameter tweaks proved insufficient in unearthing higher-value strategies. Instead these findings point to the need for more significant interventions: radically different exploration methods, more frequent or intermediate feedback signals (e.g. mini-episodes), and domain-specific state abstractions. I believe these measures could help the agent overcome the inertia of local minima and exploit more informative patterns with the goal of ultimately leading to more competitive and meaningful performance.

4.5 Conclusion

Despite the strong theoretical foundation of the Multiplicative Weights Update Method (MWUM)—rooted in game theory, optimization and machine learning—the practical application of MWUM to Hanabi underscores significant implementation challenges. Although the algorithm offers clear theoretical guarantees and has historically proven effective in regret minimization and adaptive strategy formation, the partially observable environment like Hanabi fell short of expectations.

Initial experiments produced average scores of about 1.2 to 1.3 out of 25 over 100,000

episodes. Adjusting simple parameters, such as increasing the learning rate from 0.1 to 0.5, yielded only marginal gains. Even more substantial efforts—improved state representations to reduce dimensionality, advanced reward shaping to provide more informative feedback and an adaptive learning rate to stabilize updates—resulted in plateauing performance around 1.27–1.29 points after 11,000 episodes. These modest improvements highlight the complexity of the state space and the difficulties associated with sparse, delayed rewards in Hanabi. The intricate interplay of hidden information, limited communication channels and interdependent actions creates a problem landscape where straightforward enhancements to the learning process do not easily translate into meaningful performance boosts.

Given these challenges I believe a promising avenue for future work would involve introducing "mini-episodes." By segmenting the game into smaller checkpoints—such as evaluating partial sequences of plays or rewarding incremental progress after each player's turn—the agent could receive more immediate signals of improvement. This more frequent feedback may help the agent break free from local performance plateaus and discover more effective strategies. Coupled with additional exploration mechanisms, such as further state abstractions, and refined parameter adjustments, mini-episodes could pave the way toward unlocking more of MWUM's theoretical potential.

Although the initial attempts did not achieve high-scoring strategies I believe the time spent working on these results provided valuable insights into the complexity of applying MWUM to a domain like Hanabi. The lessons learned here underscore the importance of ongoing refinement, richer intermediate feedback and careful consideration of domain-specific constraints.

5 Githubs:

Diviendu: https://github.com/divy-sh/hanabi-deck-validator
Jen and Sara: https://github.com/Jericgordon/Hanabi-Math

References

[1] Arora, S., Hazan, E., & Kale, S. (2012). The multiplicative weights update method: a meta-algorithm and applications.

[2] Bauza, A. (2010). *Hanabi* [Card game].

[3] Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. *Proceedings of the Eleventh International Conference on Machine Learning*, 157–163.

[4] Brown, G. W. (1951). Iterative solution of games by fictitious play. In *Activity Analysis of Production and Allocation*, 374–376.

[5] Hannan, J. (1957). Approximation to Bayes risk in repeated play. In *Contributions to the Theory of Games*, Vol. III, 97–139.

[6] Littlestone, N., & Warmuth, M. K. (1989). The Weighted Majority Algorithm. In *30th Annual Symposium on Foundations of Computer Science*, 256–261.

[7] Freund, Y., & Schapire, R. E. (1997). A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1), 119–139.