Agent Evaluation in Hanabi-Playing Scenarios

Martim Monis
Instituto Superior Técnico
Lisbon, Portugal
martimrosam@tecnico.ulisboa.pt

Mateus Pinho
Instituto Superior Técnico
Lisbon, Portugal
mateusleitepinho@tecnico.ulisboa.pt

Tiago Quinteiro
Instituto Superior Técnico
Lisbon, Portugal
tiagoquinteiro@tecnico.ulisboa.pt

Figure 1: A set of Hanabi playing cards.

ABSTRACT

In this study, we investigate the application of several agents in the cooperative, hidden-information card game Hanabi. We strove to replicate the agents found in this paper [1] (from which a lot of our work is based on), picking from some of the best and worst agents and evaluating them against each other.

KEYWORDS

Teamwork, Imperfect information, Card game, Cooperation, Smart agents, Advanced decision making

1 INTRODUCTION

Hanabi is a cooperative, partially-observable board game that garnered the prestigious Spiel des Jahres award for best board game of the year in 2013. This recognition has spurred its inclusion in various academic studies, as detailed below.

Several aspects make Hanabi an appealing subject for research in agent modeling. Firstly, it is a cooperative game where agents must collaborate to achieve a common objective, discouraging greedy behavior. For instance, assisting another player in scoring a point is more advantageous than playing a risky card that might end the game prematurely. Secondly, the game's rules incorporate well-defined communication actions regulated by a resource that agents must manage effectively. Finally, the presence of hidden information, where no single player can view the entire game state, adds a layer of complexity requiring intelligent reasoning about imperfect information.

Our goal with this project will be to simulate a multi-agent system, and evaluate how various approaches to agent design impact the final score of Hanabi. We also intend to investigate how agents cooperate and communicate with each other.

Section 1.1 delineates the rules of Hanabi. Section 2 details the implementation of the agents designed to play Hanabi under these rules. Section 3 outlines the testing and evaluation methodology for these agents. Section 4 presents the test results. Section 5 discusses and interprets the findings.

1.1 Hanabi

Hanabi is a cooperative game where a team of two to five players strives to complete five stacks of sequentially-numbered cards, each representing one of the game's five suits. The deck consists of 50 cards, each with a specific suit and rank. The suits are white, yellow, green, blue, and red. Each suit contains three cards of rank 1, two cards each of ranks 2, 3, and 4, and a single card of rank 5. The game also includes two types of tokens: information tokens and

life tokens. The game begins with the players collectively holding 3 life tokens and 8 information tokens.

Each player starts with a randomly dealt hand of five cards, held such that they cannot see the suit or rank of their own cards but can see those of the other players. The remaining cards form a face-down draw deck, accessed during game play.

During each turn in Hanabi, players can choose from three types of actions:

- **Tell:** Select another player and point to all of their cards that share a specific number or suit. This action costs one information token.
- **Play:** Choose a card from your own hand and play it.
- **Discard:** Choose a card from your own hand and add it to the discard pile.

In a Tell action, cards can only be identified by their suit or rank, not both simultaneously. Moreover, only existing cards can be identified; it is not allowed to inform another player that they have no cards of a particular suit or rank.

Playing a card involves adding it to the stack of its matching suit. Players do not need to know the precise stack beforehand; for instance, a player can play a card that they believe to be a 1 without knowing its exact suit at the start of the game. Each card in the stack must be of the correct suit and have a rank one higher than the card below it (with 1 cards starting new stacks). If a card is played out of sequence, the team loses one life token. Completing a stack of a given suit rewards the team with an additional information token, provided they do not already have the maximum of eight.

Discarding a card is permissible only if it can potentially restore an information token. Thus, either a Tell action or a Discard action must always be feasible. After discarding or playing a card, the player draws a new card from the draw deck. Discarded cards are visible to all players, and discarding a card increases the number of information tokens, up to the maximum.

The game proceeds until all cards in the draw deck have been drawn, at which point each player gets one final turn, and then the game ends. The game also ends if the team loses all of their life tokens.

Scoring is determined by summing the top card of each completed stack. The highest possible score in the standard game is 25, achieved by completing all five suits. Remaining life or information tokens do not contribute to the score in the standard game.

1.2 Coordination in Hanabi

In the game of Hanabi, each agent possesses unique information, rendering a centralized multi-agent planning approach impractical due to the necessity of maintaining private information.

The Tell action in Hanabi, which incurs the cost of an information token, necessitates efficient communication about a player's hand. Given the constrained set of communication actions, coordinating through direct communication presents a significant challenge, making Hanabi an intriguing subject for research.

An important aspect to consider is that Tell actions can convey implicit information: since cards of a specific suit or rank must be identified, cards that remain unidentified likely do not match the criterion. This negative information can accumulate over turns, eventually revealing a card's identity or its playability. Such knowledge is especially potent in the endgame.

Human players often enhance their use of Tell actions through conventions, selectively identifying cards as playable. For instance, if Player 2 holds (R, 1), (B, 1), ... and the table stacks are (R, 1), (B, 0), (G, 0), (W, 0), (Y, 0), Player 1 might choose to indicate the suit rather than the number to avoid highlighting the non-playable red card. Player 2 can then deduce that the identified card is likely playable, knowing Player 1 would not signal a non-playable card. Furthermore, since the suit was indicated instead of the number, Player 2 can infer they possess a non-playable 1, albeit without knowing its exact location. This sophisticated use of information necessitates an understanding of how the player will interpret and act upon the provided information in accordance with their strategy.

1.3 Implementation Details

For creating, running and interacting with Hanabi matches, we chose to use *Python* [2] with the *PettingZoo* [3] framework which provides a relatively mature interface and environment for this kind of work (it in turn, makes use of *OpenSpiel* [4] and *Gymnasium* [5]). Additionally, we utilized several of Python's available libraries such as *NumPy* [6], etc., as *PettingZoo*'s internals rely on their data structures.

Our choice of *PettingZoo* for the environments came with some unpredicted implementation quirks. One is the relatively slow performance of some of our implemented agents (the ones relying on Monte Carlos' Tree Search algorithm) due to its lack of a way for duplicating environments, forcing us to instantiate new environments and replay all previous actions whenever a copy of an existing one is needed. Another is due to discrepancies between the documentation and the code: for example, the observation vector that contains information each player can retrieve from the board on their turn is not as described in PettingZoo's webpage, having various indexes shifted. This led to a lot of sunken time debugging our program, when in fact the fault was with the framework we were using, and in the end we decided to stick with only 2 player matches as that's what we have somewhat correct documentation. In cases of 3 or more players, we determined that the observation vector shifted even more, leading to the loss of crucial information to our implementation, without which it was impossible to execute the agents' rules. This means that, to run Hanabi games of 3-5 players, we would need to implement our agents on a new environment entirely, which was unattainable in this stage of our project.

Moving on to the code structure of our project, we implemented our agents and testing framework in 5 files:

- **hanabi.py**: implemented here is the PettingZoo Hanabi environment we used to test our agents and rules throughout the development process. When beginning a game, via terminal, we choose what agents we want to run the game with from our pool of policies. Besides this, we also store an important global variable called cards_age in this file, which is updated after every turn and important to one of our rules (DiscardOldestFirst);
- **policies.py**: as the name implies, we implemented all our policies in this file. Besides MCTS, which has a relatively complex structure, every other policy has a straightforward

sequence of rules, sometimes tied together with conditions for each rule to execute;
- **rules.py**: where the rules, intricate to our policies, are implemented using the information conveyed to us by the environment's observation vector, the most important bits of data being:
  - cards in the other player's hand;
  - remaining deck size;
  - current state of the fireworks;
  - discard pile;
  - revealed info of other player's cards;
  - revealed info of current player's cards.
- **utils.py**: in this file, we implemented a Wrapper for the hanabi_v5 class, in which we could add important attributes to our implementation, such as *action_history, errors* and *seed*. It was also in **utils.py** that we defined helper functions that update the cards age and check for errors after every play;
- **tests.py**: this is where we produced all the exhaustive testing we put our agents through. We will detail our method of testing further below.

2 SYSTEM ARCHITECTURE

Several controllers utilized in this experiment were implemented as production rule agents. Many of these agents share individual rules, each described here independently. All rules include additional preconditions to ensure legality within the game rules (e.g., a Discard action requires checking for available information tokens). To maintain brevity, it is assumed that the rules of Hanabi are properly adhered to (e.g., informing a player about a card also informs them about other cards satisfying the same criterion).

- **PlaySafeCard**: Plays a card only if it is guaranteed to be playable.
- **OsawaDiscard**: Discards a card if it cannot be played by the end of the turn. This includes cards disqualified from being playable, such as a card with an unknown suit but a rank of 1 when all stacks have been started, or cards whose prerequisite cards have already been discarded.
- **TellRandomly**: Tells the next player a random fact about any card in their hand.
- **DiscardRandomly**: Randomly discards a card from the hand.
- **PlayIfCertain**: Plays a card if it is certain which card it is and that it is playable.
- **DiscardOldestFirst**: Discards the card that has been held in the hand the longest.
- **IfRule(λ) Then (Rule) Else (Rule)**: Executes the first rule if the Boolean λ expression evaluates to true; otherwise, if a second rule is provided, it executes that rule.
- **PlayProbablySafeCard (Threshold $\in$ [0, 1])**: Plays the card most likely to be playable if its probability is at least the given threshold.
- **TellDispensable**: Tells the next player with an unknown dispensable card the information needed to correctly identify it as dispensable, targeting cards that can be identified with a single piece of information.
- **TellAnyoneAboutUsefulCard**: Tells the next player with a useful card either the remaining unknown suit or rank of the card. This rule focuses first on telling the player of immediately playable cards than can be added to the firework pile. Then it tells the other player about any 5's they might have, as those cards are very rare and would be worth it not to discard. After that, it looks for cards where one tell could work for multiple cards, such as a player with three blue cards or two 4's.

2.1 Agents

On our project proposal, we set ourselves to implement a group of different agents each with a unique policy picked from the paper this work is based on. As already mentioned, most of these agents make use of a subset the rules listed previously. The initial list was as such:

- Flawed
- Legal Random
- MCS - Legal Random
- IGGI
- Piers
- Predictor IS-MCTS

However, we decided on replacing Predictor IS-MCTS with MCTS-Piers. There were many reasons that led to this decision, however the most significant one is Predictor's reliance on the randomness of other agents with which it is matched. As described on the original paper,

> *(...) The predicted agents were initialised with random seeds: this corresponds to the predictor's having knowledge of each agent's overall strategy but no knowledge of its internal workings.*

Two of the implemented rules depend on randomness, TellRandomly and DiscardRandomly, however the majority does not and instead makes use of probabilities derived from the available information to make a informed decision.

Furthermore, among the chosen agents, the ones with most significant reliance on random choices are also only two, MCS - LegalRandom and LegalRandom, which mostly make completely arbitrary legal plays, so having an understanding of their internal workings would not yield any meaningful contribution to Predictor's performance.

As such, we decided to not implement Predictor, which would also carry a great time toll on our project due to its complex nature, and instead implement MCTS-Piers. The original paper's Piers results highlight it as a promising agent, therefore we deemed it a good pick for the missing spot in our selection of agents, keeping the number of good and poor performing agents balanced.

For some extra variety, we also decided to implement MCTS-IGGI, as it was a very simple addition and could give some contrast in relation to MCTS-Piers and the IGGI policy itself.

2.1.1 Flawed. This is an agent designed to be intelligent but with some flaws: it does not possess intelligent Tell rules, and has a risky Play rule as well. Understanding this agent is the key to playing well with it, because other agents can give it the information it needs to prevent it from playing poorly. The rules used in order are

- PlaySafeCard
- PlayProbablySafeCard(0.25)
- TellRandomly
- OsawaDiscard
- DiscardOldestFirst
- DiscardRandomly

2.1.2 LegalRandom. A *Random Agent*. It uses the observation of the board to filter out illegal plays that could prematurely end the match, choosing any of the available legal actions at random during each turn.

2.1.3 MCS-LegalRandom. An upgrade to LegalRandom by combining the Monte Carlos' Tree Search algorithm with it as the policy for the rollout phase. However, it is limited to a 1 turn simulation phase (maximum depth of 1) and 1 second time limit. The MCS prefix specifies the configurations of MCTS where this simulation limit of one turn is in place.

Furthermore, it is also relevant to point out that, for any of the implemented agents utilizing the MCTS algorithm, there are some parameters to be tuned (besides the ones already mentioned). These affect the simulation phase result, calculated through the following weighted score expression,

$$\langle score,\ lifeTokens,\ remCards \rangle \cdot \vec{weights}$$

where,

score : the sum of successful plays
lifeTokens : the number of remaining life tokens
remCards : the number of remaining cards
weights=$\langle$ **10, 3, 1** $\rangle$: the weights vector

2.1.4 IGGI. A cautious and more predictable agent that avoids losing life tokens. The rules used in order are

- PlayIfCertain
- PlaySafeCard
- TellAnyoneAboutUsefulCard
- OsawaDiscard
- DiscardOldestFirst

2.1.5 Piers. This is an agent designed to use IfRules to improve the overall score. Otherwise, it is similar to IGGI. The rules used in order are:

- IfRule (lives >1 And Not deck.hasCardsLeft) Then (PlayProbablySafeCard(0.0))
- PlaySafeCard
- IfRule (lives >1) Then (PlayProbablySafeCard(0.6))
- TellAnyoneAboutUsefulCard
- IfRule (information <4) Then (TellDispensable)
- OsawaDiscard
- DiscardOldestFirst
- TellRandomly
- DiscardRandomly

The first IfRule is designed as a hail Mary in the end game: if there is nothing left to lose, try to gain a point. This derives from human play, when typically during the end game we make random plays if we know there is a playable card somewhere in our hand. This rule is more accurate, as it uses all the information it has gathered to calculate probabilities. The second IfRule simply risks playing a card if there is a reasonable chance of it being safe. The third IfRule is designed to try to provide more intelli gent Tell conditions. If there is nothing useful to Tell and we are low on information, we set another agent up to be able to discard cards that are not needed. This means that the agents can burn through cards that are not helpful so as to try to obtain useful cards from the deck.

2.1.6 MCTS-Piers. An upgrade to another agent, Piers, by combining the Monte Carlos' Tree Search algorithm with it as the policy for the rollout phase. This configuration is also limited in time and depth, but the values have been raised to 2 seconds and 4 turns, respectively.

2.1.7 MCTS-IGGI. An upgrade to the IGGI agent, that combines the Monte Carlos' Tree Search algorithm with it as the policy for the rollout phase. This configuration is limited in time and depth, with values of 2 seconds and 4 turns, respectively.

3 METHOD

3.1 Evaluation Metrics

To assess an agent's success in Hanabi, we used four metrics:

- **Final Score (0 to 25)**: as it was explained before, the final score of a game is the sum of every firework's last played card. Because achieving the highest score possible is the main focus of Hanabi, it was clear to was that we had to place the most importance on this metric.
- **Average Errors (0 to 48)**: in our analysis, we consider that an error occurred when, during the course of a game,:
  - a life token is consumed (meaning that the card played was a 1 of a suit that had been played in a previous round or or not the next number sequentially in a suit that had been played in a previous round);
  - a card that could have been played in the current or in a subsequential round is discarded;

  As the game develops, we keep a tally of how many errors an agent has made and save this value when the game comes to an end. Through this metric, we can get a sense of an agent's capability (or lack thereof) of reaching the right decision and of its cooperation abilities.
- **Number of Actions Taken (0 to ∞)**: to get a sense of how long some agents drag their games out, we also counted how many actions their games lasted. With this measure we can have a sense of how erroneous an agent is while playing Hanabi, as games end rather quickly when a player is constantly playing wrong cards and burning through the life tokens.
- **Remaining Information Tokens (0 to 8)**: an essential part of Hanabi, the information tokens spent during the course of a game are a very telling metric to understand how good our agents are at communicating with each other. There is also a fine balance between using the information tokens insightfully and resorting to them on a great deal of plays because the agent doesn't know any better, which is interesting to examine.

3.2 Testing Methodology

We tested all the agents we implemented, by pairing every agent with every agent and extracting the metrics above from every game. Every game had a random seed, and we played all agents against each other 50 times. Since we had 7 agents playing against other 7 agents 50 times, we have a total of 2450 games. Upon completion, we logged all the metrics to a file, and then aggregated them to analyze useful metrics, such as the average of every metric.

We also collected some information with a different score function for the MCTS agents (as a form of comparison) for which we did 245 games, and processed the results in the same way as above.

3.3 Validation

Before we delve into our results, we first need to check how correct our implementation is, in comparison to the results from the original paper.

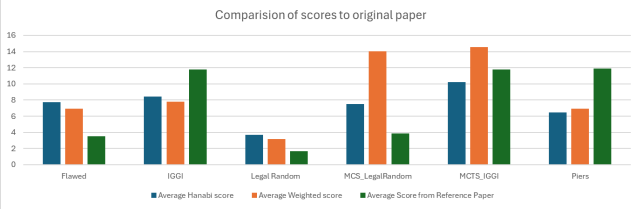

Figure 2: Comparison of scores to the original paper

As we can see from Figure 2, our scores are overall better, but they mostly line up with the reference. The reasons for our scores being better are probably, the following:

- We have a higher percentage of MCTS agents, inflating the overall score.
- We have a different score function for the MCTS agents, and they get much better results when compared to using the same function from the original paper.
- Piers and IGGI performance is probably affected due to us interpreting their description of the rules differently from what they coded.

4 RESULTS

4.1 Score-based Results

The first thing we will analyze, is the average score per each policy (Figure 3).

The scores mostly line up with what is expected, with MCTS scores being comparatively very good, and LegalRandom being expectedly bad.

Furthermore, the success of every combination of agents when playing Hanabi is also important to the average score analysis, which we can observe in Figure 4.

There, we can clearly see how much the results from Figure 3 have been inflated by the MCTS polices, and we if we split the heat map into two blocks, we can very clearly see differences.

Another interesting point of evaluation is to monitor how different the average scores are for each agent when playing with other non-MCTS agents vs when playing with other MCTS agents, described in Figure 5.

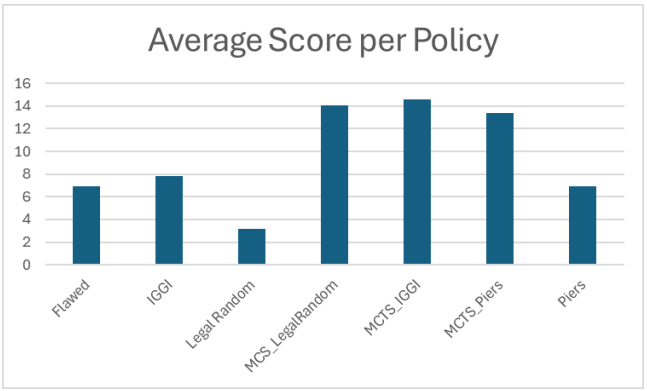

Figure 3: Average score per each policy.

Average score of Policy x Policy							
Heatmap	Flawed	IGGI	Legal Random	Piers	MCS_LegalRandom	MCTS_IGGI	MCTS_Piers
Flawed	2.66	4.24	1.96	3.48	13.38	11.36	11.3
IGGI	4.24	9.12	1.48	7.72	9.92	11.78	9.96
Legal Random	1.96	1.48	1.18	1.4	6.52	4.82	4.56
Piers	3.48	7.72	1.4	7.14	8.34	11.28	11.88
MCS_LegalRandom	13.38	9.92	6.52	8.34	20.3	20.54	20.26
MCTS_IGGI	11.36	11.78	4.82	11.28	20.54	20.3	19.36
MCTS_Piers	11.3	9.96	4.56	11.88	20.26	19.36	15.9

Figure 4: Average score for each policy when paired with every other policy.

As we already mentioned in the heatmap analysis, it is obvious that agents perform enormously better when paired with an advanced MCTS agent.

Our last graph regarding final score details the final average score of all agents when we run MCTS with the weighted average score heuristic detailed above vs when MCTS chooses which paths to take simply considering average score. (Figure 6)

From that information, we can conclude that MCTS takes better decisions when guiding itself by a weighted average score metric, taking into account score, remaining cards and life tokens.

4.2 Errors-based Results

Another important way to analyze the behaviour of these agents is by taking a look at the errors they commit during the course of a game, which is showcased in Figure 7.

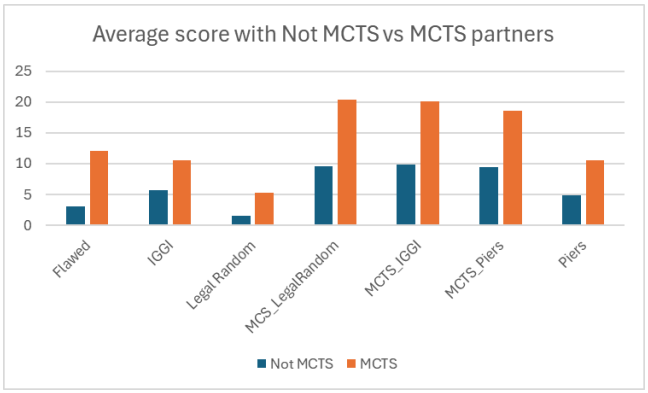

Figure 5: Average score for each agent when playing with non-MCTS agents vs MCTS agents.

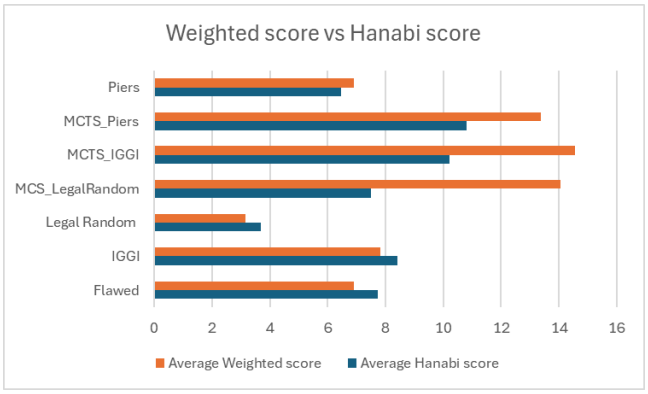

Figure 6: Average score for each agent when MCTS uses weighted average score vs average score.

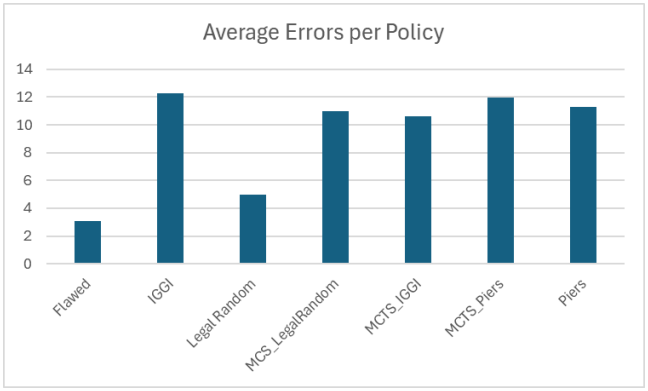

Figure 7: Average errors for each agent.

From the large number of errors made by policies that discard cards early in their sequence (such as Piers and IGGI), we can deduce that they often resort to discarding a random card too quickly.

Still regarding this matter, we thought that it would be curious to compare errors and actions per game and do a graph that details the ratio between them, to see if there is any correlation between the two metrics. It would only be logical that agents whose games last longer put themselves in a position to collect more errors. (Figure 8).

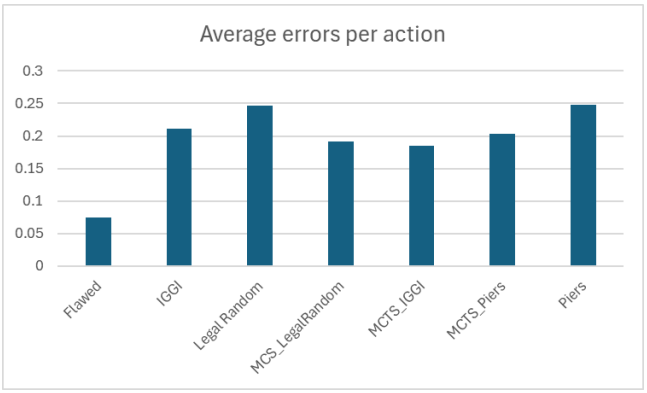

Figure 8: Average errors per action for each agent.

We find that, besides the Flawed policy that doesn't make errors because it rarely discards cards, the other agents are fairly consistent in these values, ranging from 0.18 to 0.24. It is fairly reasonable to assume that there is some correlation between actions per game and errors per game.

4.3 Actions-based Results

In this section, we present a graph which examines the length that an agent takes to finish a game, on average. (Figure 9)

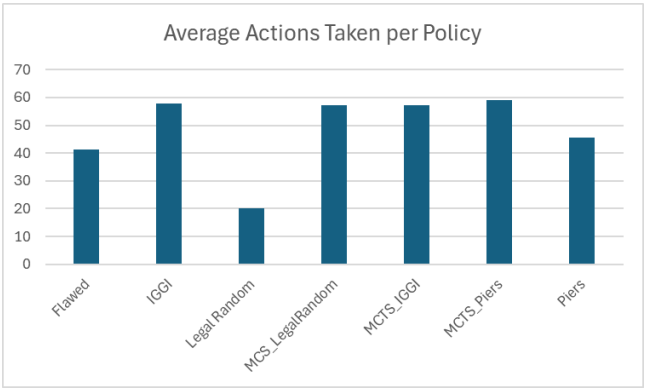

Figure 9: Average number of actions for each agent.

Examining the results of the graph above, we can see that our least intelligent agent - LegalRandom - also takes the least actions, because it spends all life tokens very early on. Conversely, all MCTS agents exhaust almost every possibility to carry the game forward and therefore take close to 60 actions to finish the games.

4.4 Information Token Results

Our last graph (Figure 10) isn't particularly interesting to analyze but we'll include it in our paper nonetheless. It regards how many information tokens remained on average in the games of each agent.

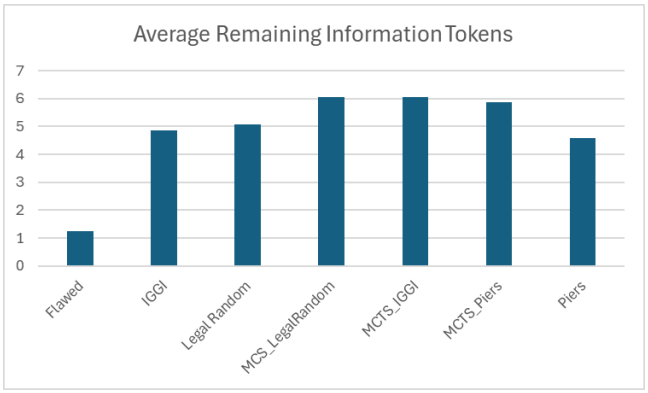

Figure 10: Average number of remaining information tokens per agent.

By reviewing this data, we can gather that the policy Flawed relies on spending information tokens far too often, and most likely with no specific logic, as the rule TellRandom is third in its sequence. On the other hand, we feel that the other policies perhaps weren't using their info tokens often enough, and might benefit from rules that are more adamant about spending them.

5 DISCUSSION

5.1 Preeminance of MCTS-Based Policies

In our view, the aspect of the results that stands out the most is how fitting it is to use Tree Search Algorithms in turn-based card games like Hanabi, MCTS in particular because of its availability in providing an answer for the next move, even when stopped. This proves the conclusions of the original paper's authors.

According to our data, it's overwhelmingly clear that agents achieve higher success when combined with MCTS than on their own. In theory, it's obvious that an agent will be stronger when given the ability to retrace its steps and try another move but it's noteworthy how much more accomplished a game is when its two participants are controlled by MCTS-based policies, constantly getting 20 plus final scores, as opposed to the contrary (two agents of regular policies) of sub 5 final scores in most games, as we can see on the heatmap (figure 4).

Of course, inside each of the 2 categories of policies there still is a great deal of heterogeneity, especially when looking at all metrics we tracked, and those differences will still be explored later, but the pivotal conclusion we arrive at is how much more productive policies are when coupled with the MCTS algorithm.

5.2 Boost of Agents' Scores

Comparing to the original paper's results (figure 2), we see that our score results are a little inflated. This has a very simple and logical explanation: in the original paper, for testing purposes, they didn't use every pairing combination possible of the pool of agents, like we did. All of their agents were only paired with a smaller fraction of said agents, which did not include ones that use MCTS.

In our research, because we wanted to further determine the impact of MCTS agents in Hanabi, we tested every possible pairing combination of our agents, which means that they were also paired with agents that rely on MCTS. Naturally, this will lead to a higher percentage of games where the agents are playing with an advanced and more intelligent agent, capable of collecting a higher score at the end. Overall, this had a big impact on the final results, just as we were aiming towards.

For example, if we consider the average final score of the Flawed policy, we can see an increase of almost two points compared to the original paper's score. Well, in our opinion, this might not stem particularly from a different implementation of the policy that gives it a greater capacity to score higher but from the agents it is playing against, a big part of them using MCTS and much more intelligent, contributing to a higher score of every agent.

On the opposite side of the spectrum, in our paper we also have a larger ratio of obtuse illogical agents (1/7 vs 1/11) - technically both researches feature only one of these unintelligent agents, LegalRandom but the original paper has more overall agents. Using the logic above, we could also argue that having a larger ratio of random agents would drag the overall average score down. However, because of the larger number and impact of MCTS agents, we feel that their influence on the overall final scores vastly outweighs the influence of the LegalRandom agent.

5.3 Analysis of the Errors

Another useful and interesting metric present in our final results is the Error count, which denunciates agents that are either moronic when it comes to their play and regularly consume life tokens or that resort to discarding cards far too quickly. In figure (7), we can see that the regular policies with the higher average error count are Piers and IGGI. If we take a look at their action sequence, we see that they both discard the oldest card first before telling a random hint (contrarily to the Flawed policy, and has very few errors because of it). We think that this behaviour is premature and may discard important cards to complete the fireworks later, especially because the highest ranked cards tend to be the oldest ones in the hand of a player. In a future attempt to implement other agents, we feel that it may be useful to loosen up the definition of a useful card (in the TellAnyoneAboutUsefulCard rule) in an attempt to prevent these unfortunate discards.

However, it's important mentioning that, in this metric, agents that carry their game out for a great number of actions are penalized, as the length of a game permits the agent to progressively rack up more errors as the game goes on. This is why the LegalRandom policy has such a low error average: a game that involves them is over 50% shorter than any other (figure 9). This also explains why this metric correlates negatively to the average score result - the longer a game is, the higher score it reaches but the more errors its players collect.

5.4 Weighted Score vs Average Score on MCTS implementation

Comparing between these two different heuristics (figure 6) ran with the MCTS algorithm was a great source of curiosity during the development of this project, as we wanted to further expand the work of the original paper and we felt that tinkering with MCTS might be a good way to improve the success of our agents. In this

sense, we're happy to report that, when MCTS is using the weighted average heuristic and taking into account not only score but also remaining life tokens and cards in deck, all our agents MCTS agents score a lot higher than they would when only concerned with final score.

This shows that it is beneficial to our agents when the final score isn't the only metric powering the execution of a MCTS and leads us into a debate of which other weights and metrics would be useful to include in other MCTS implementations. Perhaps if we had given a higher weight to the remaining life tokens or had also considered the remaining info tokens, these agents would have achieved even higher scores. In any case, we believe that this data is some of the most relevant in our research and that it is of great interest to further develop in future investigation.

6 CONCLUSION

In conclusion, we verified that the Monte Carlo Tree Search algorithm vastly improves results in the game of Hanabi, and devised a new score function that allows it to achieve even higher scores in comparison to [1]. These results are consistent with the findings of [1], which state MCTS' importance.

REFERENCES

[1] Joseph Walton-Rivers, Piers Williams, Richard Bartle, Diego Perez Liebana, and Simon Lucas. Evaluating and modelling hanabi-playing agents. pages 1382–1389, 06 2017.
[2] Python. Python, 2024.
[3] Farama Foundation. Pettingzoo, 2024.
[4] Deepmind Technologies. Openspiel, 2023.
[5] Farama Foundation. Gymnasium, 2023.
[6] NumPy. Numpy, 2024.