Evaluating MCTS in a New AI Framework for Hearts

Wells L. Santo and Anthony Wong
Computer Science and Engineering Department, New York University, New York, NY
{wl.santo, aw1818}@nyu.edu

Abstract — Although much work in AI for Games today is focused around the digital medium, there is still much to be said about the medium of traditional card games. In particular, this paper attempts to look at the characteristics of imperfect information games, by focusing on a case study on the trick-taking, imperfect information card game Hearts. In performing this research, the authors of this paper designed an entire Hearts framework from scratch—one that could properly model imperfect information while allowing for AI agents to be easily developed and tested. Finally, after designing a working framework that models imperfect information in Hearts to allow for AI agents to perform lookahead computations, we created four algorithmically distinct agents to play Hearts. These included a low-playing decision tree player, a random-playing decision tree player, a greedy lookahead player, and the headlining (UCT) MCTS player. By designing these AI, we were able to find that given our representation of imperfect information, both the MCTS player and the greedy lookahead player were still capable of playing well, as well as interestingly, even with our given limitations.

1 Introduction

Because of the immense profitability and growth of the digital game-playing industry in the last few decades, it is quite sensible to see a dramatic growth in research and development on artificial intelligence for very specialized digital video games. Whether it be for basic platformers, first-person shooters (FPS), or more daringly, real-time strategy games (RTS), much work has been put into develop better (and more human-like) non-playable characters for the future. However, this research takes the study of artificial intelligence for games in a completely different direction—by looking back into the past with the tried-and-true medium of traditional card games. More specifically, this research aims to look at the card game of Hearts, which has one major characteristic that makes it important to study even in a modern digital gaming context—the issue of imperfect information.

What it means for a game to have imperfect information is to allow for players to have private knowledge that cannot be seen by others. In sharp contrast to games like Chess or Checkers, where all the positions of each player’s pieces are known at all times, Hearts presents the challenge of imperfect information by giving players no information about the hands of their opponents. Imperfect information is also a much better model of many other games, including Poker and even Starcraft, where Fog of War becomes an issue. By selecting a better model for AI agents to work with, we believe that more interesting and fair forms of intelligence can be developed. Of course, this not only levels the playing field for human and AI players, but also presents a new problem of dealing with how to model the AI systems for such imperfect information games anew.

In addition, in order to make this problem even more interesting and focused, Hearts was also chosen for its classification as a “trick-taking” card game, which gives it very structured and specific properties that make it easier to design certain AI agents (such as the Monte Carlo Tree Search) for. In a trick-taking card game, individual cards are played at every discrete round, with a termination after a finite (and often minimal) number of rounds. These games are also often scored based on the number of rounds (or tricks) won, which provides a very strong initial heuristic to base player decisions off. Our primary research question was whether using the MCTS algorithm for a trick-taking card game could do well, even in the face of imperfect information.

Naturally, this research does not occur in isolation. In section 1.1, we discuss the related research and motivations that have driven this research, before diving into the specific details of our work. Since our readers also may not be completely familiar with the game of Hearts, an explanation of its gameplay and rules are briefly discussed in section 1.2. In section 2, we will describe the implementation of our Hearts framework, as well as the crucial design choices that were necessary to make in order to support lookahead playouts of the game while staying true to the imperfect information model. Section 3 then discusses four AI agents that were then developed for this framework, including the Monte Carlo Tree Search (MCTS) agent that headlines this paper. Section 4 lists the results of these agents when put into competition with one another, and Section 5 concludes by discussing lessons learned and potential future work on this topic.

1.1 Motivation and Related Research

Initially, this project was intended to explore the gameplay of the card game Napoleon, which just like Hearts, is an imperfect-information trick-taking card game. However, since Napoleon is a much lesser
known game that is also far more intricate than Hearts, the project was modified to select a game that is more popular and more easy to be modelled. Of course, based on the similarities between Hearts and Napoleon, we selected Hearts as our representative, since it is extremely well-known here in the United States. We also decided to stick with the paradigm of trick-taking card games, because of the fact that we felt it would be easier and more interesting to model for, as compared to other imperfect information card games such as Poker.

One paper that motivated us to select trick-taking card games was written by Bonnet, Jamain, and Saffidine, who showed that perfect information trick-taking card games (such as a perfect information version of Hearts) was PSPACE-Complete when designing a “pure winning strategy,” while assuming (but not proving) that designing for imperfect information trick-taking card games was NP-Complete.\[1\]

In doing further research for this project, we found similar work done by Buro, Long, Furtak, and Sturtevant from the University of Alberta, who decided to pursue the same problem, but with the German card game Skat.\[2\] In their research, they were able to develop a Skat player “that plays at the level of human experts,” by using state evaluation techniques fueled by immense amounts of domain-specific knowledge and collected game data. In their closing remarks, they noted that they were “fortunate to have access to a large volume of human game data,” which they used to train their player. In their work, they were able to develop an “imperfect information game player and show that training on data from average human players can result in expert-level playing strength.” Of course, our project deviated dramatically from theirs in three regards: (1) since we do not have access to such vast data, we could not develop a player that could train off actual games of Hearts, (2) we found that the emphasis on such domain-specific techniques were not too interesting and instead opted to investigate the more general intelligence MCTS algorithm, and (3) we wanted to develop our own framework with our own models for imperfect information, as opposed to using our player to play physical, in-person games, as they did.

In designing AI players for an imperfect information game, we also did consider related studies in card games like Poker, such as the Billings, Davidson, Schaeffer, and Szafron paper, which discusses the importance of imperfect information modeling, as well as the lack of a good Poker AI in 2002.\[3\] They noted that when compared to perfect information games, “there are fundamental differences in the structure of imperfect information game trees, and the total number of possibilities to consider is prohibitive,” which for our purposes, made using MCTS as our headlining algorithm all the more appealing.

The final push that would motivate us to working with Hearts, and MCTS specifically, were two papers written by Jeff Rollason of the commercial company AI Factory. Since our team was familiar with the Hearts app released by AI Factory (which we heartily play on long subway rides), we were happy to see his work in moving away from evaluation-based algorithms and instead exploring imperfect information trick-taking card games using MCTS. In his work for the card game Spades, Rollason noted that the MCTS player was “well ahead” of the evaluation-based player, but with the trade-off that “most moves were actually worse.”\[4\] Here, Rollason seemed to favor the state evaluation agent, which a few of members of our team agreed with. After all, expert human players operate only on evaluation-based grounds, without random lookahead, and they are able to do it quite well. Thus, we were compelled to try our own at designing a few different agents to compare to MCTS in the game of Hearts.

As a corollary, we also found it extremely interesting to think about how to model imperfect information in the game of Hearts, as we needed to design our own framework to support AI players. Again, Rollason’s work was a crucial motivator, as another one of his papers discussed exactly the issue of dealing with imperfect information in game playouts for MCTS.\[5\] Here, Rollason pitches the very poignant question: “How can you search when you do not know what is going on?” to which he solves by “‘guess[ing]’ who might have which card.” Though this leads him to a conclusion in developing the “Information Set MCTS” (ISMCTS) algorithm, we decided to go in a completely different direction and avoid such guessing by modeling imperfect information using randomness.

1.2 The Game of Hearts

Because Hearts is an imperfect-information trick-taking card game, it is not only fun to play, but quite
difficult to play as well. The popularity of Hearts can be captured with the anecdote that the game is shipped natively with every copy of the Windows operating system. (Indeed, that is how most of us on the team first came to hear about the game.) Like other trick-taking card games, many variations of Hearts are available, with the most standard ones involving four players. For this project, we will look at the standard version of Hearts, which involves four players and uses a standard 52-card deck without Jokers. In this game, cards are also given ranks as follows, in increasing order: 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K, and A.

In every game, each player is dealt a 13-card hand, and must select one card to play every round. Thus, each round consists of four total cards having been played, with 13 rounds total per game. The player who holds the Two of Spades must start the game off, and the taker of each round starts the next round. Based on the cards that the taker collects for that round, the taker will gain points: 1 point for each Hearts card and 13 points for the Queen of Spades. However, the objective of the game is to gain the least amount of points, not the most.

Of course, there are rules for what cards can be played each round. If cards have already been played by other players in that round, the player must play a card that is the same suit as the suit of the first card played that round. If the player does not have a card of that suit, he/she may play any card from his/her hand. The taker of each round is the one who plays the highest rank card within the suit of the first card played that round.

The player who starts the round off is not limited to the same restriction, and can choose to lead with any card from his/her hand, with a caveat. The first time that a Heart has been played in the game, we follow the terminology that “Hearts has broken,” and if Hearts has not broken in the game so far, no one can lead with a Hearts or the Queen of Spades, unless those are the only cards remaining in their hand.

Typically, players of Hearts will play consecutive games until any one player achieves over 100 points. At that time, the player with the lowest amount of points received wins. In our implementation of the game, we will not follow this rule, but rather, we will play through a series of 10 games per set and check who has the lowest score. This is to allow more games to be played out, since each game will typically result in 26 points being distributed among the players.

Finally, one of the most interesting aspects of Hearts is the “Shooting the Moon” rule, which occurs when one player takes control of the game and takes all 26 points in a single round. When this occurs, the player will actually take 0 points in that round, while the three other players will each take 26 points. This introduces an element of deception into the game, where players must bluff their way from a losing position to the best possible position.

Though Hearts by itself may be an easy and fun game to follow, the strategies involved in playing it may become extremely complex. Bluffing, counting cards, and prediction can all give a player an edge over the others, which is exactly what makes this game so interesting to play and develop intelligent agents for. Indeed, this game also lends itself nicely to the development of many diverse forms of AI, which may appeal to different gameplay styles or understandings of the game. For our purposes, the use of score in Hearts primarily helps us to design a heuristic for our AI agents, while the existence of imperfect information is what makes the design of our framework quite interesting, which we explain in the following section.

2 The Framework

Many considerations went into the designing of our Hearts framework. For one, we needed something that could easily hook in AI agents for quick prototyping, development, and deployment. For another, we needed a way to model game states so that we could retain the imperfect information characteristics of Hearts. In the end, by basing much of our design choices off of the work of the GVG-AI Competition framework, we were able to very nicely accomplish both goals in one fell swoop. Because we previously had worked with the GVG-AI framework, which is implemented in Java, and we have existing MCTS players already designed for GVG-AI, we found it best to also create this framework in Java as well. This ended up being to our extreme benefit, as Java’s use of pointers and classes quickly became the supporting grounding that would make the functionality of our game design choices possible. Note that further documentation on our framework can be found in (excruciating) detail on our public GitHub repository, which is linked at the end of this paper.

2.1 The Card and Deck Classes

To begin our framework, we started out with the essential building blocks of getting a card game to function—the cards and the deck. Our first design choice was choosing how to represent individual cards, and whether they should be distinct objects at all. (After all, how would we then deal them out from the Deck, while maintaining both the Deck and the players’ hands?) In the end, we found that there were enough unique functions that Cards needed to have, so we designed a full class to model each Card. We also decided to stick with good Java practices, and defined two sets of enums, one for Suits and one for the Value that each card can have. Card objects would thus have two member variables—one for its Suit and one for its Value, with the standard compareTo() and equals() methods overloaded to do shallow compares between cards. We also made our Card class slightly fancier by...
print out the unicode symbols for each of the suits instead of printing the names of the suits themselves. This was done elegantly using a simple switch statement, as follows:

```java
switch(suit) {
    case SPADES: shorthand += "\u2660"; break;
    case HEARTS: shorthand += "\u2661"; break;
    case DIAMONDS: shorthand += "\u2662"; break;
    case CLUBS: shorthand += "\u2663"; break;
}
```

In the Deck class, we would store all 52 cards in two separate ArrayLists—one for pointers to Cards current in the Deck (called allCards), and one for pointers to Cards that are in the hands of the players (called invertDeck). Every time a player would play a card, we would invoke a function within the Deck class that would move the pointer of the Card out of invertDeck and back into allCards. The code for this was simple, as follows:

```java
void restockDeck(Card returned) {
    allCards.add(returned);
    invertDeck.remove(returned);
}
```

With this implementation, it became extremely easy to keep track of the cards that have already been played in any game, which would become important later on in simulating full and random playouts of the game. Furthermore, this implementation would allow us to restock the deck automatically, without extra effort between games, and allow us to re-deal each player’s hands efficiently at the beginning of each game.

Other interesting methods that we implemented as part of the Deck class include the initDeck() method, which would make use of our Suit and Value enums to generate all 52 cards in merely three lines of code, and the shuffleDeck() method, which makes use of the very helpful Collections.shuffle() function that Java already provides to shuffle elements in ArrayLists.

### 2.2 The Game Class

What drives each game is of course the Game class, which uses the playNewGame() function to run through all 13 rounds of a full game of Hearts for each of the four players. This class is essentially the heart and soul of all the Hearts-specific gameplay, including the function checkRound() to see if the specific card played that round was valid and the function findTaker(), which goes through the cards played in each round to see who wins the round and receives the points. As part of our implementation choice, we decided to decouple every component of the game quite thoroughly, so that all functions could be easily debugged and used as separate and independent modules. Points of interest in this class include having ArrayList<Player> playerOrder to keep track of all four players, ArrayList<Card> currentRound to keep track of the cards currently played this round, boolean hasHeartsBroken to flag whether hearts has broken in the game, and ArrayList<Integer> playerScores to keep track of the current score of each player within this game (this is distinct from the score that each player keeps track of, which is discussed in the following subsection). By setting up the Game in this way, it became very obvious how to hook in each of the four players, how to keep track of the Deck, and how to deal with the State class for independent playouts of the game.

Of course, since this is the primary class to drive the actual rules and structure of the game, most of the functions here are hard-coded with conditionals and loops based on what needs to happen in order for a game of Hearts to be properly played. It may also be important to note that in our implementation of Hearts, we did implement the functionality of shooting the moon. However, we ignored one aspect of the full game of Hearts—the “card passing” stage that occurs at the beginning of each game. We decided that the question of card passing would be too different from our main goals of the project, and decided to leave it out of this iteration of the framework to simplify our research. Given future work, this is something that would be added to the Game class.

### 2.3 The Player Class

Although very straightforward, the Player class is an abstract class that forms the foundation for implementing all AI players. Each Player object will keep track of three key pieces of information: their name, the total amount of points that they have received between all games, and an ArrayList<Card> for their hand. Otherwise, most of the functions of this class are simply getters and setters to retrieve the name of the player, add points to the player, add cards to the hand, check for the existence of specific suits in the hand, and print the player’s hand. One interesting addition to the Player class that we decided to add at the last minute is the SuitRange class, which actually goes through the hand and keeps statistics on the range of indices corresponding to specific suits. Finally, to bring it all together, two abstract methods are defined for the Player, which must be implemented by all derived classes—setDebug() and performAction(). The functionality of each method is described in the code snippet below:

```java
abstract boolean setDebug();
abstract boolean performAction (State masterCopy);
```

// Return true if you want your player to print debug messages

// Given any sort of player, make a decision to play a card
// Pass in a copy of the game state for full playout functionality,
// The player must return with the Card they choose to play in
// that round
2.4 The State Class

Perhaps the most important piece in our entire framework, the State class is what allows for full playouts of each game on the Player side, which required specific design choices in dealing with the imperfect information characteristics of Hearts. At every round, a State object is passed into each player’s performAction() method, for the player to use in making his/her/its decision. Full playouts to simulate the rest of the game are achieved by calling advance() this State object, before returning the final and official selected Card to play for that round.

First and foremost, we do not keep track of the hands of any of the players in the State class. Because players do not have knowledge of the cards that others are holding, we decided that it would clearly be a violation of the imperfect information characteristic for the State class to retain this information. Thus, in full playouts of the game, no sense of “who played what” is implemented into our framework, since players in real games of Hearts cannot accurately guess which players have which cards in their hands.

Of course, this is a disputed topic, as some expert players may argue that they can look at the playing history of each player and infer what cards that player is holding. We respond by saying that since our State does keep track of what cards have already been played, and since our State does give the Player information about what cards are played during full playouts, this sort of inference is something that should be implemented by specific AI agents, and not given as a resource for all agents to have access to.

What we do keep track of, however, are (1) the cards that have been played, (2) the cards currently played this round, (3) whether hearts has broken in this game, and quite interestingly, (4) the scores of each of the players. We found it important for players to know what the scores of others looks like, so that they can base playing decisions off of this information. If players have a specific penchant for wanting to give points to the player with the most points, for instance, this information would definitely prove to be valuable. Essentially, these are the only pieces of information that we provide each player in evaluating the current state of the game.

Finally, the crowning gem of our framework comes in the advance() function of this class. This is where the “random playout” of our game occurs, such that a full game of Hearts can be simulated while respecting the limitations of imperfect information. Once the player calls the advance() function, we will first check that the play is valid before finishing up the round and starting the new round up until the point the player must make a decision again. During this time, since one round has ended, this function will also do computations to check who won that round, how many points have been earned, and who will lead in the next round. This information is then used to update all of the member variables of the State, including the simulated Deck, which keeps track of cards that have been played, as well as cards that have yet to be played. This function will also return the number of points that this specific move has cost the player (if the player won that round), or the value -1, to denote that an invalid play has been performed. Essentially, this function is the very engine that allows simulated gameplay to proceed, providing any AI agent enough resources to make decisions with.

The most important thing to notice about the advance() function is how playouts for each round are performed. Instead of making predictions on who will play what card, and instead of limiting play to cards that are within the suit of the first card played in each round, we will do purely random plays in each round. This is because there is no guarantee that a certain player still has cards of any given suit, and may be completely valid in playing an off-suit selection. Because we keep track of all cards that have not yet been played in the invertDeck member of the Deck class, selecting a valid card to play at random is not very difficult at all. Indeed, the rest of the functionality of the State class reduces to just basic bookkeeping to make sure that the State is updated properly after every card is played, so that valid rule-following playouts are performed within the confines of the imperfect information constraints. Below is a code snippet of the main section that deals with the playout for a single round of the game.

```java
// Check if this game is still going
if (isGameValid()) {
    // Repeat until it's the player's move again
    int index = rng.nextInt(cardsPlayed.invertDeck.size());
    // Make sure that card is not in this player's hand
    while (isInMyHand(cardsPlayed.invertDeck.get(index), playoutHand)) {
        index = rng.nextInt(cardsPlayed.invertDeck.size());
    }
    // Use the play card method to put take the card out of the
    // invert deck, into the played deck, and also onto the table
    playCard(cardsPlayed.invertDeck.get(index));
    taker = (taker+1) % playerScores.size();
}
```

3 Four AI Agents

For this project, four AI agents were designed to play Hearts using our framework: (a) LowPlayAI, (b) RandomPlayAI, (c) LookAheadPlayer, and (d) MCTSPlayer. In each following subsection, we will simply analyze and discuss some of the design and implementation choices for each of the four agents. The results of how each agent ended up playing can furthermore be found in section 4. Note that none of our players were explicitly designed with the mindset
of shooting the moon, and there may still be many far more interesting agents that can be developed for our Hearts framework.

3.1 LowPlayAI

Since Hearts is a game where a player only gains points from playing the highest card in the round, it made sense to program a very simple decision-tree-based agent that always played the lowest valid card possible. The decision-tree logic of this agent is visualized in Figure 2. This player is extremely interesting primarily in the fact that it has the potential to play really well while being extremely simple—after all, it will never play a card to win a round unless it has absolutely no other options. This seems to model the gameplay of most beginner Hearts players quite accurately, without introducing much complexity at all into the implementation.

Of course, one important caveat with the implementation of this agent is how it selects the “lowest” card in the hand. Although it is sensible to discuss the lowest card in any given suit, if the agent runs into a situation where it is not forced to play within one suit, there is no official ranking for the four suits themselves. And yet, we were able to keep the implementation of this agent to be extremely simple by following the conventions of our framework. Because players’ hands are always kept sorted in the order of Clubs, Diamonds, Spades, and Hearts, by selecting the card with the lowest index to play, this will guarantee that the Two of Clubs is played first (if this player has it) and that Hearts are not played until the end of the game.

3.2 RandomPlayAI

In order to provide a sort of benchmark for the other AI agents, this RandomPlayAI was developed. Like the LowPlayAI, the logic behind this agent is also quite simple and can be visualized in decision-tree form, as seen in Figure 3. Briefly speaking, this player will play from random any card that is valid this round. If it starts the round, it will still select an opening card at random, keeping in mind whether or not Hearts has broken in this game.

3.3 LookAheadPlayer

This is the first AI agent that makes use of the “random playout” functionality of our Hearts framework. This player is slightly greedy in the sense that it will do a single random playout for each valid card each round, and select to play the card whose playout returns the lowest amount of points accrued. This is an interesting agent only in the sense that it is completely based on how closely our framework models the imperfect information characteristic of the game. Given a deterministic, perfect information game, this can be seen to be similar to a Max-Max sort of algorithm, which looks to shallowly select a next move which is maximal in its returns.

This agent also fares well in the trick-taking card game paradigm, since we can directly evaluate how “good” a move is based on the points achieved at the
end of the game, which will only be 13 moves away at most. Since the depth of the game is not very deep, and this controller only does playouts for any of the valid cards in the hand at the moment, at the absolute worst it can only iterate through a very loose upper bound of 169 possible moves total. In any ordinary game, where the valid cards to be played in each round is limited by suit, this agent will never actually come close to reaching this upper bound. Below is some pseudo-code which describes the implementation of this agent in more precise detail.

Algorithm 1 LookAheadPlayer
1: procedure performAction(State s)
2: bestPoints ← 0
3: bestCard ← validRange[0]
4: for Card c in validRange
5:     cs ← s
6:     points ← cs.advance(c)
7:     points += playoutGame(cs)
8:     if points > bestPoints
9:         bestPoints ← points
10:         bestCard ← c
11: return bestCard
12:
13: procedure playoutGame(State s)
14: totalPoints ← 0
15: while s.isGameValid()
16:     c ← validRange[randomIndex]
17:     totalPoints += s.advance(c)
18: return totalPoints

3.4 MCTSPlayer

Last but not least, an MCTS player was designed to play Hearts within our framework. This MCTS player can essentially be classified as running a plain UCT algorithm, as it makes no optimizations in playing the game and does not use any heuristics to expand the tree, beyond looking at the final score after a full random playout of the game. The expansion policy and tree policy of this MCTS player follow the “general MCTS approach” as described by Browne, Lucas, and Tavener.\(^7\) Otherwise, this MCTS player uses the Upper Confidence Bound for Trees (UCT) algorithm in order to decide which parts of the tree to expand at any given moment, and selects to play the card that returns the maximal rewards as a result of running the UCT MCTS algorithm.

Beyond following the specifics of the conventional UCT MCTS algorithm, there are only a few minor changes that we have made for the algorithm to work in the setting of Hearts. To begin with, since there are a restricted set of moves that a player can make at any given time, we restrict the tree policy of our algorithm to only expanding the tree to nodes that represent valid moves. Naturally, this can be seen as a type of pruning that improves the runtime of our algorithm, but otherwise, it is not conceptually more optimal in gameplay (since non-valid cards cannot be played anyways). During the random playouts of the MCTS default policy, our algorithm will play random, but valid, cards, much like the RandomPlayAI, all the way through a complete game, since a game can only last at most 13 rounds. This is computationally efficient, and we found that Hearts really lent itself nicely to the structure of the MCTS algorithm. At the end of a random playout, the points that the MCTS player received is subtracted from 26 (the maximum number of points that can be achieved in any game), and propagated back up the nodes of the tree, as in a standard MCTS implementation. In this sense, the “best reward” is measured by how effectively the MCTS player was able to play when minimizing the points accrued per simulated game. Again, no other heuristics were used to bias the MCTS player into making its decisions.

4 Results

So how well did our MCTS algorithm stack up against the RandomPlayAI benchmark, as well as the other AI agents? In order to measure the success of each algorithm, we first had each agent play ten sets of ten games (or one hundred games total) against three RandomPlayAI agents, in order to have some standard benchmarks to base future findings off of. Of course, with any good experiment, we conducted a control group test as well, in which four RandomPlayAI were pitted against each another for ten sets of ten as well. This control group test was meant to show that the element of chance in any one game of Hearts (based off a random dealing of cards in the deck) would not create enough “noise” to bias any one player into winning in the long-term. This control group test also empirically showed that the player ordering does not matter (and since cards are randomly dealt, with the first play going to the player with the Two of Clubs, it should not matter in theory).

As seen in Figure 4 on the following page, the random playing agents were typically evenly good, with all of them receiving around 70 points per set of ten, with the potential of sometimes achieving over 100 points. One interesting thing to note in the control group test was that out of these 100 games, Random Player #1 actually was able to Shoot the Moon once in set 6. Overall, however, this did not dramatically change the distribution of points at the end of the ten sets, and as we will note later on, this was not too uncommon of an occurrence of Shooting the Moon.

To jump to the results of how our MCTS algorithm fared against all of the other agents, we actually ended up playing ten sets of ten with each of the four AI agents in the game at the same time. The results of this can be found in Figure 5 on the following page. Not surprisingly, MCTS quite handily defeated both the
RandomPlayAI and LowPlayAI agents, with quite noticeable margins of difference. However, what was extremely interesting about our result was the effectiveness of the LookAheadPlayer, which shockingly not only defeated the RandomPlayAI and the LowPlayAI as well, but was able to square off (and slightly surpass) the MCTS agent.

In a round-by-round analysis of the game, we found that the MCTS player would make extremely clever, human-like moves, where it would play high if it felt like it would not take the trick that round, or play low if it did. This is potentially a consequence of the fact that if a player holds on to a high card for too long, it may end up gaining points quickly in the late game, which is commonly known among intermediate players of Hearts. Yet, because of the constraint of how imperfect information was modelled in our framework, the MCTS player could only foresee random play-throughs of each game with each tree traversal, which led it to still be capable of playing bad moves. This, of course, is exactly the case with the game of Hearts when played by human players.

The LookAheadPlayer was more interesting, as it played moves that were often counterintuitive—it would play neither the highest or lowest card of a given suit when an ordinary human player would have made that move. Perhaps by giving it the power of random playout lookahead, this agent was able to have deeper insight into the card that would uniquely give it the least amount of points given the current hand (quite greedy, in a sense) and select that card.

But we were not convinced that a four-point margin meant definitive victory of the LookAheadPlayer over the MCTS agent. We continued our experiments by doing the benchmark tests for the LowPlayAI, the LookAheadPlayer, and the MCTSPlayer, as seen in Figure 6. These games actually ended up reflecting the results of Figure 5 quite well, in the sense that the LowPlayAI still did better than the RandomPlayAI, but...
not by much, while the LookAheadPlayer and the MCTSPlayer continued to easily defeat the Random-PlayAI. Though it was mildly interesting to empirically verify that a truly greedy low-play algorithm would not do much better than a random-play algorithm, it was once again shocking to see the LookAheadPlayer do better than the MCTSPlayer. While the MCTSPlayer was able to finish with a modest score of 512 (149 points less than the best RandomPlayAI), the LookAheadPlayer completely dominated with a final score of 436 (a whooping 272 points less than the best RandomPlayAI). Of course, this is not to say that the MCTSPlayer did poorly. On the contrary, out of all sets of ten games that we simulated, the MCTS player was able to achieve the absolute best score of 12 points in that set, which not even the LookAheadPlayer (or most of the authors of this paper) could manage.

We then decided to turn up the heat on the MCTS player by pitting it against a wave of three LowPlayAI instead of RandomPlayAI. Again, it won handily, with a score of 460 (231 points less than the best LowPlayAI). This can be seen in Figure 7. From these results (which we arrived at after exposing MCTS to 300 games of Hearts) we concluded that MCTS did indeed perform well, despite the constraint of imperfect information. With additional tuning, optimizations, and domain-specific heuristics for evaluation, we predict that MCTS could easily become a very expert and human-like player within our Hearts framework.

Of course, this still did not settle whether the LookAheadPlayer or MCTSPlayer was the best in its current implementation. We played a final ten sets of ten between two LookAheadPlayer agents and two MCTSPlayer agents to settle the dispute once and for all… and the LookAheadPlayer team definitively won. With a combined score of 1268, the LookAheadPlayer agents nicely took down the MCTSPlayer agents, who scored a combination of 1384 points. This was a margin of victory of 4.37% of the total 2652 points that were earned within these 100 games. Notice that though 100 games normally end in 2600 total points, the extra 52 points occurred when the first MCTS agent actually shot the moon in set 10. Though we did not explicitly set up any heuristics for giving extra value to shooting the moon in the MCTS algorithm, shooting the moon does still result in a 0-point game, which is still optimal to the MCTS player. The final results of these ten sets of ten can be seen in Figure 8.

5 Conclusion and Future Work

In conclusion, in this paper we were able to discuss the development of a full Hearts framework that is capable of modeling imperfect information, such that artificial intelligence agents are able to easily hook into the framework and make decisions based on random playouts. We also discussed the design choices of four different AI players, two of which made immediate decisions based on a decision tree, one of which made use of the random playout functionality in a greedy manner, and one of which ran our headlining MCTS algorithm. However, as the results stacked up, it seemed that the LookAheadPlayer was able to marginally defeat the MCTSPlayer in a series of experiments, which leads us to believe that there may be something valuable behind the use of the LookAheadPlayer. This may also be a factor of the lack of heuristics and tuning used in our MCTS algorithm, which may allow it to be moderately effective in a general sense, but not so much in a domain-specific setting.

Regardless, it seems that both the MCTS and LookAheadPlayers do well, even given the constraints of imperfect information, and the fact that we do not boost either with much domain specific knowledge beyond minimizing the amount of points earned per game. As shown in Section 4 of our paper, a plethora of empirical evidence supports the claim that these two general agents can easily surpass dummy decision-tree or basic finite-state-machine agents that do use domain-specific knowledge to operate. Because these quite general players are able to do extremely well, and even mimic intermediate human gameplay of Hearts,
we found that this project was quite the success, and that further work on this framework and on a more diverse set of AI agents may be fruitful.

In the future, it may also be valuable to optimize the MCTS agent and tune for better heuristics that may make it better in general for any sort of trick-taking imperfect information card game. Though it may be very interesting to see how that agent would do when applied to frameworks that play similar card games like Spades, Napoleon, or Skat, there are still several interesting things that can be done within our own Hearts framework, as well. For instance, it may be interesting to create an alternate State class that models having perfect information in the game, so that we can compare the effectiveness of MCTS with our current imperfect information constraints. It may also be valuable to update our representation of imperfect information (which currently reduces the problem to purely random playouts of the remaining cards) to something that may potentially be a better model for both Hearts and generalized imperfect information games.

Of course, while much more research and work would have to be put into the framework, we are excited to have developed for this project a fully functional Hearts playing framework, which is implemented for quick development and testing of new AI agents. With this framework, we can not only allow many human players to play against a diverse set of AI opponents, but we can also open the doors for other programmers to design their own AI agents to compete with others in Hearts. What may eventually be very interesting is if this framework could be “crowd-sourced” and used to host AI development competitions, so that many different individuals could develop their own AI and help advance our research in creating agents for imperfect information games. The framework would only need very minor adjustments to deal with this amount of variation of AI agents and automate gameplay between them.

For further development by others, our entire framework is also completely open-sourced and public, as found on GitHub in the following repository: https://github.com/Devking/HeartsAI

Acknowledgements

The authors would very much like to thank New York University (namely the Polytechnic School of Engineering) and Professor Julian Togelius for offering the special topics course CS9223 Artificial Intelligence for Games at the last minute in Spring 2015, which facilitated the execution of this project. Without the weekly guidance of Professor Togelius and the Teaching Assistant, Edward Melcer, this project would not have been as well-formed and interesting as it ended up becoming. Of course, designing the Hearts framework for this project could not have been at all possible without insight into the GVG-AI Competition Framework, which is currently run by Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon Lucas.

References