Multi-Ply Planning for Duelyst

CS 730: Intro to AI
Final Project Report

Bryan McKenney

Problem

Duelyst II (henceforth referred to simply as "Duelyst") is a two-player online collectible card game (CCG) that features a 9x5 grid board where unit cards can move around and fight. Each player's goal is to keep their General alive and kill the enemy General using the resources in their deck.

Normally, players in Duelyst play against each other, which provides for exciting gameplay. However, there is also a human vs AI mode which can only be used to train new players because the AI, called the starter AI, is weak against anyone who knows the game, leading to boring gameplay. It is also designed to play only the starter decks and cannot do much more than that. I want to create an AI that can consistently beat the starter AI and that can use any deck in hopes that it will allow more experienced players to have worthy battles. Dream Sloth Games, the developers of Duelyst, have allowed me to work in their codebase for my research, and if I create something that works well enough, perhaps it will get into the game. This opportunity is also exciting because there is only one research paper relating to Duelyst — for matchmaking, not game-playing AI (Cowan 2023) — and few about CCGs in general, so it is a new domain to explore. Smart AI, assuming it was fast enough to play against humans, could open up a host of options for Duelyst. For instance, there is a "sandbox" mode where a player can test two of their decks against each other, but they have to play both decks themselves. I believe it would be more interesting to fight a strong AI. There used to be boss battles in the game and there are plans for a roguelike mode (where the player battles through AI encounters), both of which could benefit from better AI. Currently, fighting the AI gives no rewards because it is too easy, but a strong AI could even be used in ranked games when few humans are online to speed up matchmaking times.

Creating an AI for Duelyst is a challenge because there is a large branching factor, several actions can be taken per turn within a 90-second limit, actions can be stochastic, and the opponent's deck and hand are unknown — it is a complex partially-observable stochastic game (POSG). As part of my thesis work, I designed and implemented some algorithms for Duelyst, including a modified version of Monte Carlo tree search (MCTS). These algorithms do not model hidden state and so can only look ahead until the end of the current turn (or ply), which is very limiting. Although MCTS and another of my algorithms perform well against the starter AI, I still doubt that they would be a match for human players. In this project, I extend my thesis work by exploring multi-ply planners for Duelyst to see how they fare against the starter AI and how they compare to my single-ply planners.

This project relates to the AI class because we discussed planning in partially-observable domains and game-playing algorithms such as minimax and Monte Carlo tree search, but we did not get to implement any game AI.

Previous Work

This section describes some algorithms for solving games and relevant applications of them.

Minimax

Minimax is the classic algorithm for solving fully-observable zero-sum games. It constructs the entire game tree, with one player playing optimally to maximize reward and the other playing optimally to minimize it. The sub-optimal version cuts off the search at a certain depth and applies a static evaluator to estimate the value of the leaf-node states. A variant of minimax called *expectiminimax* has chance nodes, which take the mean of their children rather than the min or max, and this allows it to play stochastic games. However, minimax cannot deal with partially-observable games.

PIMC

Perfect-information Monte Carlo (PIMC) search suboptimally solves POSGs by sampling possible worlds from a belief state and doing minimax in each one to find the best action on average across the sampled worlds (Furtak and Buro 2013).

This approach has been applied to standard-deck card games such as Skat (Furtak and Buro 2013) and also makes up an important part of Ginsberg (2011)'s GIB algorithm for Bridge.

MCTS

Monte Carlo tree search (MCTS) is an algorithm that has been used to great success in fully-observable stochastic

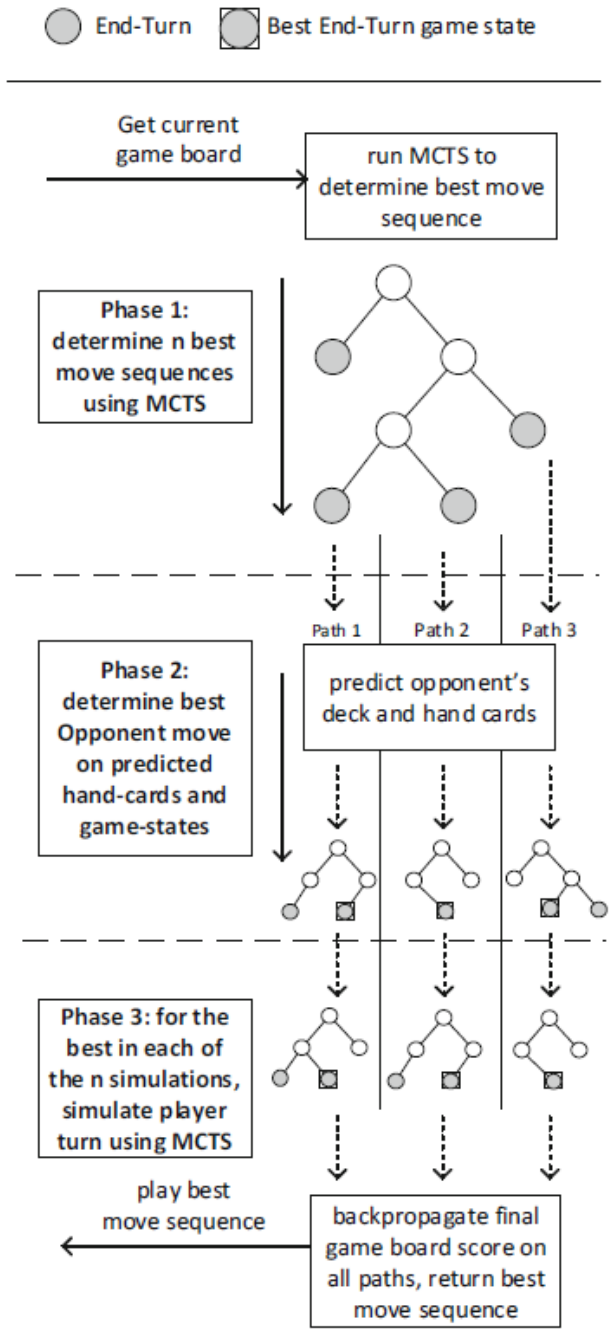

Figure 1: The modified MCTS used for *Hearthstone* by Dockhorn et al. (2018). Image from Dockhorn et al. (2018).

games (Gelly et al. 2012). It builds a search tree of alternating state and action nodes, each of which contain a visitation count N and an estimated value V (initially both set to 0). Repeated simulations are performed from the root of the search tree and one node is added to the tree each time. A rollout is done from each added node until a terminal state and then the estimated value of the node is backpropagated up the tree. This process is continued until time is up or a certain number of simulations is run, and then the action with the highest value or visitation count is taken. The UCT policy selects which action to take during simulations, and it incorporates the visitation count to add an exploration bonus (weighted by the parameter C) to nodes that have not been explored much.

Dockhorn et al. (2018) use a modified version of MCTS (see Figure 1) to play *Hearthstone*, a popular CCG that is very similar to Duelyst, except that it does not have the grid board. They run MCTS for the agent, rolling out until the end of the turn instead of the end of the game, then select the n best end-of-turn nodes reached during rollouts and run MCTS on those for the opponent, then repeat this process to simulate the agent's next turn from the n best ending opponent states. In order to simulate the opponent, they predict that the opponent's deck is one of 20 preset options (whichever is closest to what they have played) and choose a hand from that using a bigram model of card co-occurrences. In this way, they are able to look ahead 3 plies while planning, though only for a few simulations. The evaluations of the final states (3 plies ahead) are backpropagated to the current-ply search tree. The algorithm returns the best sequence of actions from the search tree (until the frontier) instead of just the best next action to take.

POMCP

Partially observable Monte-Carlo planning (POMCP) is an algorithm invented by Silver and Veness (2010) to apply MCTS in partially-observable domains. The search tree alternates between *observation* and action nodes. Observation nodes contain unweighted particle belief states that are updated during simulations. A simulation starts by sampling a state from the root's belief state so as to continue simulating from a fully observable state.

Dobre and Lascarides (2018) use POMCP to play *Settlers of Catan*, a complex 4-player POSG. They cluster actions into types — such as road building, city building, and trading — and learn a preference distribution over those types from game data. This distribution is used to influence which actions get selected during tree search (the policy is PUCT rather than UCT) and rollouts (an action type is sampled first, then an action from that type).

Background

This section describes Duelyst, my MCTS-based algorithm for playing it, the static evaluator I use, and the performance of the algorithms I developed for my thesis.

Duelyst

In a Duelyst match, two players do battle with 39-card decks on a 9x5 grid board. There is a maximum hand size of 6.

Each player controls a General and their goal is to kill the opponent's General (by reducing that General's Health to 0). To do this, they can play minion, spell, and artifact cards from their hand by spending mana. Minions are played on the board and can move and attack enemy units (enemy units are the opponent's minions and General). Spells have an effect when played. Artifacts boost the power of a player's General until they are destroyed (they have three durability and lose one every time the General is hurt).

Duelyst has a discrete but very large state space. There are 6 factions, and each faction (except Vetruvian) has 34 cards — 3 artifacts, 14 minions, and 17 spells (Vetruvian has 18 spells). There are 112 neutral cards, all minions. A player's deck can be made up of cards from one faction and neutral cards and can contain up to 3 copies of the same card, which means there are $\binom{(34+112)\times 3}{39} \approx 8.93 \times 10^{55}$ possible decks for a given faction. There are also 2 tiles, which go under minions on spaces, and 26 token cards that can only be created during matches (not added to decks). 45 units and tiles can be on the board at once, 6 cards in each player's hand, and potentially more than 39 cards in each player's deck (because some cards add cards to a player's deck). Units on the board can have Attack ranging from 0 to 99 and Health ranging from 1 to 99 (both integers) and can also have an unlimited number of additional abilities granted to them by cards or lose their abilities entirely. All of this makes the state space enormous.

Duelyst has a discrete action space that can be relatively small or very large depending on how many cards are in the player's hand and how many units are on the board. It is not uncommon to have a branching factor of over 100 at some points during the game. In addition, several actions can be taken in a turn within a 90-second time limit and some actions have stochastic outcomes.

One-ply MCTS for CCGs

My algorithm for Duelyst is called *One-ply MCTS for CCGs*. It is MCTS but with three major changes:

1. Rollouts only go until the end of the turn instead of the end of the game, because the opponent's deck and hand are unknown. Rollouts either take random actions or do hill-climbing, greedily choosing the action that leads to the highest-valued state (according to the static evaluator).
2. Inspired by Dobre and Lascarides (2018)'s use of action types to guide search, action types are an explicit part of the MCTS search tree — the possible actions at a particular state are grouped into an *action tree*. Duelyst's action space can be neatly factored into seven action types: Mulligan, MoveUnit, AttackWithUnit, PlayCard, ReplaceCard, EndTurn, and Followup (a Followup action can occur after a PlayCard action or another Followup action). For the MoveUnit, AttackWithUnit, and PlayCard action types, they can be further factored (in the next layer of the action tree) into *action sources*, or the unit or card that you are going to perform that action with (e.g. MoveUnit could branch into UnitA and UnitB). Underneath action types or action sources are ground actions, which are fully described actions that can be executed in the state (e.g. where to move UnitB). The EndTurn action type is unique in also being a ground action. Each action type and action source node has a visitation count and stored value, just like the other nodes in the search tree, and choosing an action consists of first choosing an action type and then potentially an action source.
3. There is some code specific to dealing with unique situations that arise from the Mulligan and EndTurn action types.

There are parameters for the rollout policy to use (random or greedy), the time to spend searching for each action, and whether or not to use action trees, as well as the default MCTS exploration parameter C. I use a C value of 10 for all experiments.

Static Evaluator

The static evaluator I created estimates the value of a state using a set of simple rules. A win state has a value of 10,000 multiplied by the remaining health of the winner (in order to differentiate end-game states), a lose state is the same but with a negative value, and any other state's value is the difference in values of both players' units. A unit's value is based off of its cost, Attack, and Health, with a penalty for being far from mana orbs (collectible resources that start on the board at the beginning of the game) and a bonus for being near enemy units (especially the General) that it can kill. A General's Health is valued at three times that of a minion to promote the ultimate objectives of staying alive and killing the enemy General.

Thesis Algorithm Performance

Alongside One-Ply MCTS for CCGs (MCTS), I tested an agent that takes random actions (Random), an agent that takes the best action based on a 1-step-lookahead using the static evaluator (Hillclimb), and an agent that performs a single rollout (until end of turn) from the state resulting after each action and then chooses the action that leads to the highest-scored state (Rollout). Figure 2 shows the winrate of these algorithms against the starter AI after 1,000 games each. The word in parenthesis after the name of an algorithm refers to the rollout policy used. The Rollout planner using greedy rollouts and the MCTS planner using either rollout policy achieve over 60% winrate, with the simple Rollout (Greedy) method actually performing the best, with a 68% winrate. That is the number that I desired to beat in this project.

Approach

In efforts to beat my previous results, I implemented a basic belief state, variants of MCTS and Rollout that rollout multiple plies instead of just one, a new rollout policy, and two new algorithms. I shall explain each of these in turn.

Belief State

Inferring the opponent's deck and hand (the partially-observable elements of Duelyst) is important to be able to

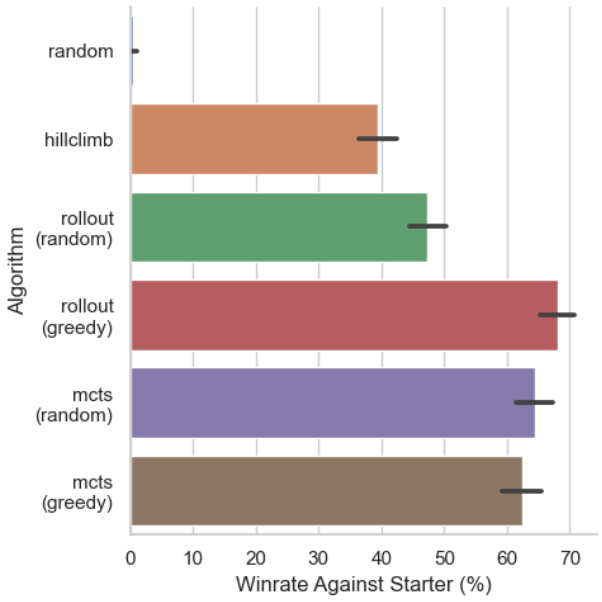

Figure 2: The winrates of the algorithms I developed for my thesis against the starter AI.

simulate the opponent for multi-ply algorithms. That is, unless the AI cheats and sees the opponent's actual deck and hand. I mostly stuck with the latter method for this project in order to determine if multi-ply search can even be theoretically helpful in Duelyst, but I also implemented a simple deck-and-hand modeling approach. It takes the current observation of how many cards are in the opponent's deck and hand, generates a 39-card deck of random cards from the opponent's faction, draws from it to construct the hand, and then removes cards from or adds cards to the deck at random until it is the correct size.

This method can be used with an unweighted particle belief state to sample potential decks and hands for the opponent.

Multi-Ply Rollouts

I wanted to see if a simpler multi-ply approach than Dockhorn et al. (2018)'s multi-level MCTS could work, so I gave the MCTS and Rollout algorithms the ability to explore multiple plies into the future with their rollouts. They can either use the random deck modeling method described above or see the opponent's actual deck in order to simulate the opponent's turns. In the former case, a deck is sampled every time the rollout enters the opponent's turn and it is applied to replace their actual deck during the rollout.

In the code and the plots, MCTS using multi-ply rollouts is still called "mcts" but Rollout is called "rolloutMultiPly."

Multi-Random Rollout Policy

In an effort to make a more robust version of random rollouts that is faster than greedy rollouts to better rollout into the opponent's turn, I devised multi-random rollouts. This policy performs n random rollouts from a state and then chooses the highest-valued (from the current player's perspective) end-of-turn state discovered to either return the value of or continue into the next ply from. A two-ply multi-random rollout with $n = 3$ would perform six random rollouts: three on the player's turn and three on the opponent's turn. This approach takes inspiration from minimax, as it chooses the best for each player and ignores lesser alternatives.

In the code and the plots, this rollout policy is called "multiRand."

PIMC

I implemented PIMC search. It takes a number of samples, a horizon, and a deck modeling method as parameters. When planning, it samples worlds using the deck modeling method and performs a form of depth-limited minimax in each. Since Duelyst turns generally consist of 5-6 actions and because the branching factor can be so large, it would be too expensive to even search through the player's turn until the first action of the opponent using minimax, so the minimax is "faked." That is, it pretends that turns only consist of one action and imagines the player and opponent going back and forth like that. Even though this seems like a strange approach, I wanted to try it because it makes the player think about the opponent's actions right away instead of at some far-off future point, and when playing Duelyst, I often think about what the opponent could do to counter a specific action that I am considering without playing out my entire turn in my head first.

In the code and the plots, PIMC is inaccurately called "minimax" because the way I use it in the experiment is essentially as minimax (since I use the actual deck and 1 sample).

Three-Ply MCTS for CCGs

I converted my One-Ply MCTS for CCGs into *Three-Ply MCTS for CCGs* by following Dockhorn et al. (2018)'s approach. One difference is that mine only returns a single action instead of a sequence of actions, as I do not see how a sequence can be returned when actions can have random outcomes, unless it only goes until the first random action. This algorithm runs three sets of MCTS (one search for the agent's current ply, n searches for the opponent's next ply, and n^2 searches for the agent's next ply). In each search, the n best end-of-turn states encountered in rollouts and the frontier nodes those rollouts originated from are tracked. These stored states and nodes allow for continuing the search into the next ply after it is over even if there are no end-of-turn states in the search tree, and for backpropagation of values through each previous-ply tree.

Each MCTS is given equal time to build a tree (which is a parameter). This is a very slow algorithm, but the point is not to be fast (yet) but to be better at playing the game.

It makes sense to continue simulating into the next ply from only the few best (according to the current player) end-of-turn states because it is similar to the way minimax works — ignoring worse options that likely will not be chosen and focusing on the best for each player. Since the additional search in some parts of the search tree goes to three plies,

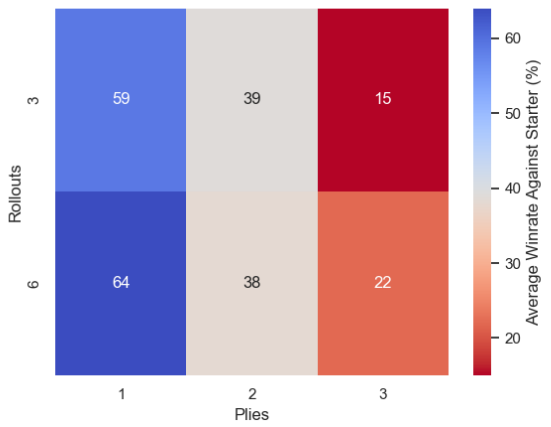

Figure 3: A heatmap of Rollout's average winrate against the starter AI using different numbers of rollouts and plies for the multi-random rollouts.

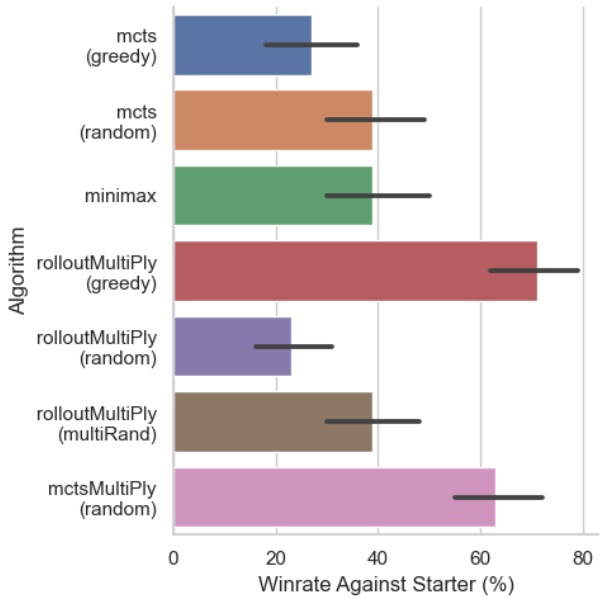

Figure 4: The winrate of each algorithm against the starter AI.

the returned values should be comparable to others in the tree because they are all from the end of the player's turn (some are just more informed than others). If the extra search only went one additional ply, those returned values would likely look worse than everything else and potentially bar the player from their best actions.

In the code and in the plots, Three-Ply MCTS for CCGs is called "mctsMultiPly."

Evaluation

I ran three experiments to judge the efficacy of my new algorithms and approaches. The first was to test Rollout with the multi-random policy with different numbers of samples and plies. The second was to compare all of the algorithms. The third was to find how impactful having an accurate deck model was on the best-performing algorithm. I will now discuss these experiments in more detail.

Rollout + Multi-Random

I tested Rollout with multi-random rollouts (and actual deck model) with all combinations of 3 and 6 rollouts and 1, 2, and 3 plies to see the effect looking further ahead had on winrate. Each combination played 100 games against the starter AI and the average winrates are shown in Figure 3. While the rollout policy appears very strong for single-ply planning, especially with more rollouts, it gets significantly weaker with each additional ply that the rollouts go out to.

Algorithm Comparison

I played each algorithm 100 times against the starter AI and recorded the winrates in Figure 4 and the average selection times in Figure 5. All of these algorithms use the actual deck model and go to two plies except Three-Ply MCTS, which naturally goes to three plies. Three-Ply MCTS planned from the 2 best end-turn-states in each search tree, had 10 seconds to build each tree, and used random rollouts. Both versions of MCTS with multi-ply rollouts had 60 seconds to choose an action. Rollout (MultiRand) was using 3 rollouts and PIMC ("minimax") had 1 sample and a horizon of 2. The black bars are 95% confidence intervals on the mean. All of the approaches do poorly except Three-Ply MCTS, which has a ∼62% winrate, and 2-Ply Rollout (Greedy), which has a ∼71% winrate. Both of these algorithms take 70 seconds on average to choose an action, which is far too long for an actual game. Three-Ply MCTS, while maintaining good winrate, did not improve over the single-ply versions shown in Figure 2. The same goes for the 2-Ply Rollout (Greedy) algorithm — while it did beat the 68% winrate of the single-ply version, this data is only from 100 games instead of 1,000, so it is likely that the winrates are about the same. MCTS with 2-ply rollouts and 2-Ply Rollout (with non-greedy rollouts) perform much worse than when only rolling out to the end of the current ply.

PIMC ("minimax"), while not boasting an impressive winrate, showed some interesting behavior that is worth looking further into. As seen in Figure 6, it lasts significantly longer than every other algorithm against the starter AI, even though it loses 60% of the time. Also, a result I found in my thesis that it appears that going second gives the player an advantage does not apply to PIMC, even though it seems to be true of every other algorithm in this experiment (aside from Rollout with multi-random rollouts, which has essentially equal winrate going first or second), as seen in Figure 7. When going first, PIMC has an average winrate of 50%, and when going second, 30%. It is not surprising that PIMC plays in a unique way given that it is a very different approach from all the other algorithms.

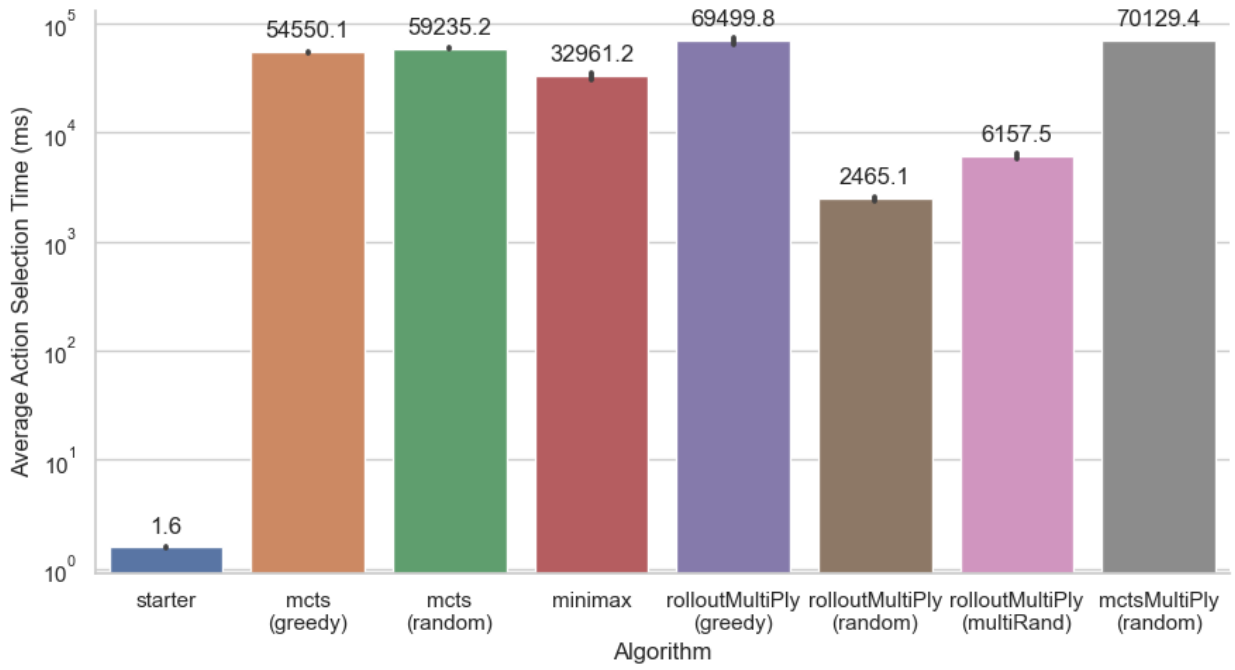

Figure 5: The average time (ms) each algorithm took to pick an action.

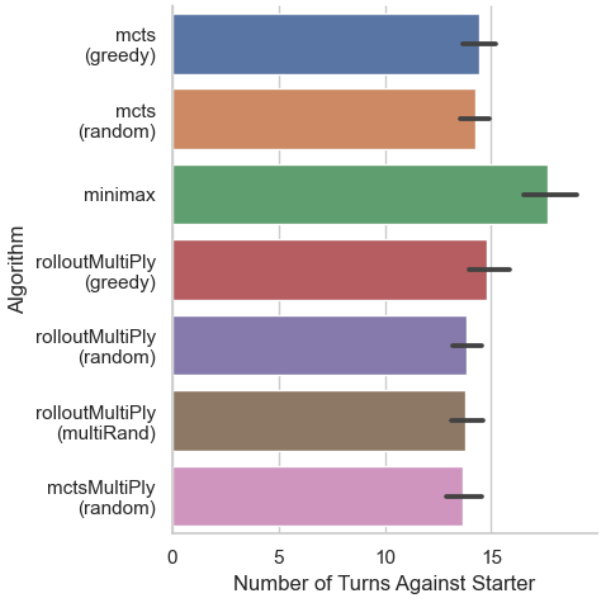

Figure 6: The number of turns each algorithm lasted against the starter AI.

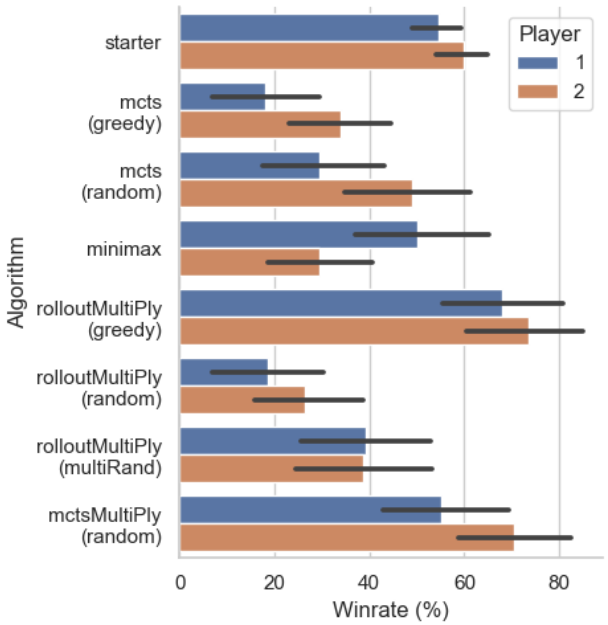

Figure 7: The winrate of each algorithm playing first and second player.

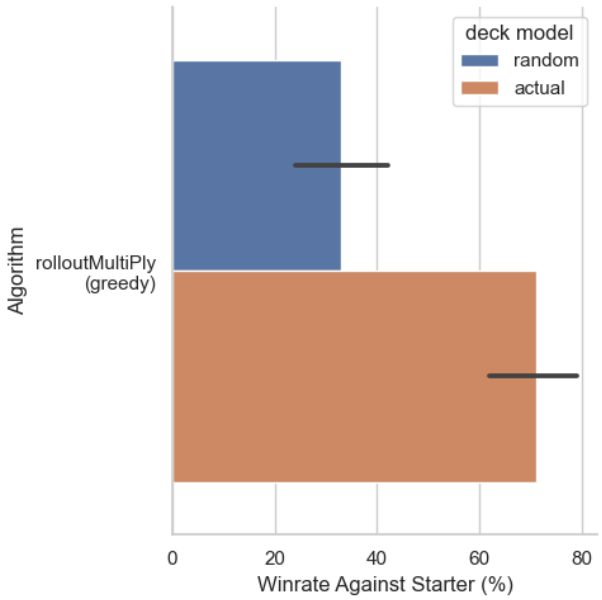

Figure 8: The winrate of 2-Ply Rollout (Greedy) against the starter AI with both deck models.

Deck Model Comparison

I took the best algorithm from the previous experiment — 2-Ply Rollout (Greedy) — and ran it 100 times against the starter AI using the random deck modeling strategy instead of seeing the opponent's actual deck and hand. The results are shown in Figure 8. As expected, it performed considerably worse, getting only around a 30% winrate. This shows that better belief states are important, but even with the best belief state — the truth — this approach and the others do not improve on single-ply algorithms and take longer to choose actions.

Future Work

Clearly, there is much work to be done. These results are disappointing because they do not show any evidence that planning ahead more than one ply is beneficial, and in many cases it seems detrimental. As an experienced Duelyst player, I do believe that thinking about the opponent's moves is often necessary to win, so the problem here could be with the static evaluator not being expressive enough to provide good information about far-future actions. I had troubles with the static evaluator when initially working on this project, and adding additional values for good minion positioning greatly improved my winrate for all algorithms (except Random, of course). It could be that I need to further enhance it to make multi-ply planning useful.

Aside from that, I want to explore the interesting results I obtained with PIMC and perhaps try other unconventional approaches to simulating the opponent.

Assuming that I can improve my static evaluator or create an algorithm that benefits from multi-ply search, I need to implement a better deck-and-hand modeling method than just randomly sampling faction cards. Randomly sampling cards from categories such as their cost and type could allow the construction of a reasonable random deck, which could have the following attributes:

1. The deck contains no more than 3 copies of any one card.
2. 60% of the deck is minions, 30% spells, and 10% artifacts.
3. The deck has a good/standard *mana curve*, which is the distribution of mana costs of all cards in it.
4. There are more faction minions than neutral minions.

I would expect something like this to perform much better than a completely random deck that may not even be possible to build, given the 3-copy limit in deckbuilding.

The next step would be predicting the opponent's deck and hand based on the cards that they have played so far, as Dockhorn et al. (2018) did. There are deck archetypes in Duelyst, and an experienced human player can figure out what deck they are playing against after only seeing a few of the opponent's cards (unfortunately, creative decks are rare, especially in higher ranks).

For any future students who want to work on an existing game like this, my advice would be to create your own simplified version of it first. I ran into numerous issues with Duelyst's codebase that took a while to overcome, and an issue of slow state copying still plagues me. However, it *is* cool to work in a real game's code. The open-source *Duelyst* code is available here (https://github.com/open-duelyst/duelyst) for anyone interested, although to work on *Duelyst II* you have to get permission from Dream Sloth Games like I did.

References

Cowan, A. 2023. Paired comparisons for games of chance.

Dobre, M.; and Lascarides, A. 2018. POMCP with Human Preferences in Settlers of Catan. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Dockhorn, A.; Frick, M.; Akkaya, Ü.; and Kruse, R. 2018. Predicting Opponent Moves for Improving Hearthstone AI. In *Information Processing and Management of Uncertainty in Knowledge-Based Systems. Theory and Foundations*.

Furtak, T.; and Buro, M. 2013. Recursive Monte Carlo search for imperfect information games. In *Proceedings of CIG-13)*.

Gelly, S.; Kocsis, L.; Schoenauer, M.; Sebag, M.; Silver, D.; Szepesvári, C.; and Teytaud, O. 2012. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM*.

Ginsberg, M. L. 2011. GIB: Imperfect Information in a Computationally Challenging Game. *CoRR*.

Silver, D.; and Veness, J. 2010. Monte-Carlo Planning in Large POMDPs. In *Proceedings of NIPS-10*.