Riichi Mahjong AI: Japanese Gambling on the Deep Web

GitHub: https://github.com/warppoint42/Mahjong221

Introduction

Mahjong is a tile-based game analogous to the card game gin rummy in which players hold a hidden hand of 13 tiles, while the remaining tiles are placed in a face-down wall formation. In each turn, they draw a tile from the wall (the equivalence of a card deck) and choose to discard one tile, aiming to form a winning hand. Tiles come in three numbered suits and honor tiles and can form melds, combinations of three consecutively numbered tiles from the same suit or three or four-of-a-kinds. A complete hand typically consists of four melds and a pair, however special winning hands are also defined, each with varying score values (see [3] for examples). Under certain criterion, players may also steal tiles as their opponents discard them to complete a meld (and reveal that meld to opponents) or win the game.

In the field of AI, Mahjong remains a game that has yet to be "cracked". As mahjong tiles in the wall or in opponents hands are hidden save for discards and revealed melds, running simple probabilistic calculations is impossible, and simulations prove difficult given a rapidly exploding state space based on unknown tiles. With three other unknown players actively building their own hands or seeking to disrupt other's hands, the social dynamic of the game proves difficult to quantify.

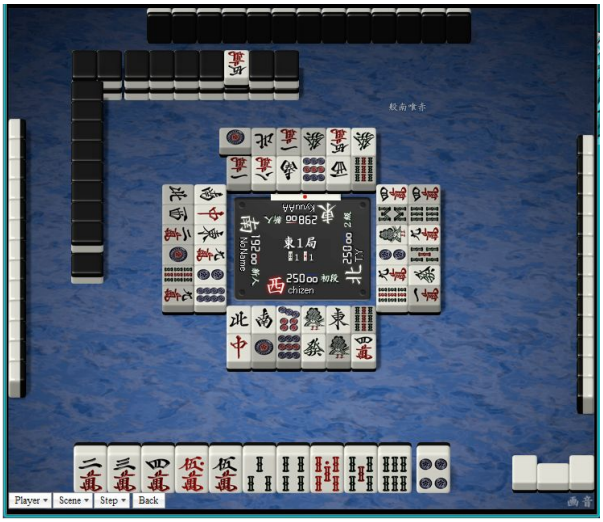
With these factors in mind, we set out to create a viable Mahjong AI player using our limited resources and algorithms taught in class, tailoring them to the complex nature of the game.

Related Work

There is surprisingly little precedent for the development of a mahjong AI or analysis of the underlying mathematics. After AlphaGo's victory, there were calls for DeepMind to challenge top mahjong players, but no response has materialized. A past CS229 project [4] focused entirely on building an hand evaluator; we chose to use

statistical values in our implementations. A 2015 study from the University of Tokyo [5] built a study that modeled and analyzed opponent's playstyles using Monte Carlo simulations, but they had a large dataset of professional matches to draw upon. In a sense, our AI is built from the ground up, as few similar implementations exist.

Platform



Tenhou.net

Our implementation runs on an online mahjong site, Tenhou.net, which uses a japanese variant of mahjong called riichi mahjong, known for its nuanced rules and win conditions. We connected to this site using the Python Mahjong package [1] and an API to interface with the website [2]. The platform matches the AI to random human players, determines draws, lists all legal moves, and handle the scoring after each winning hand.

The main challenges with the platform includes having no access to play logs (which rules out supervised methods), the random players matching (which rules out adversarial methods), a 11-30 minute game play time (which is too long to sufficiently train learning algorithms), 10 second decision time limit (which limits the complexity of the algorithm), and IP banning after multiple games.

To overcome these limitations, our implementations were be limited to search and simulation based algorithms, and games were played over the Tor network to enable playing of simultaneous games.

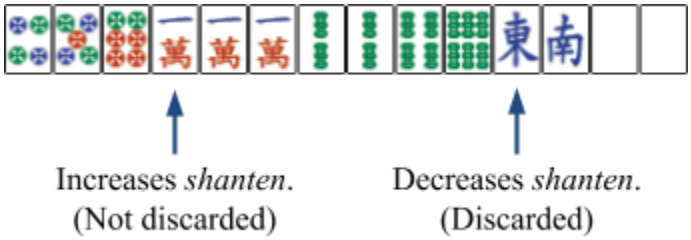
Task Definition and Approach

Given the many nuances of Riichi Mahjong, we decided to focus entirely on optimizing hand win rate, favoring completion of any hand over completing hands of maximum value or avoiding losses. The reasoning for this was that winning many small value hands should be able to stand up to winning few large value hands, especially on an unranked server versus opponents of mixed skill. In addition, the vast majority of the player's influence in the game is through choosing what tiles to discard and when to call melds, as these are the only two major contributors towards improving a hand. As such, our AI implementations largely focus around evaluating tiles to discard to build winning hands as fast as possible.

As mahjong tiles in the wall or in opponents hands are hidden, a game state space largely consists entirely of the tiles in the player's hand and the location of all revealed tiles. Assuming that the locations of the unknown tiles are random, the state space is still enormous, with up to 36 different tiles to be drawn and 14 tiles to be discarded on a player's turn, with each of four players taking up to 37 turns. To simplify and prioritize states, or discarding certain cards, we can calculate a statistic called *shanten*, or the minimum number of tile draws required to reach *tenpai*, a hand one away from winning. Assuming that all tiles are available, a lower *shanten* indicates a strictly better hand. Using *shanten* and the randomness assumption, we can begin to make state-based search models to form the base of he AI.

Models

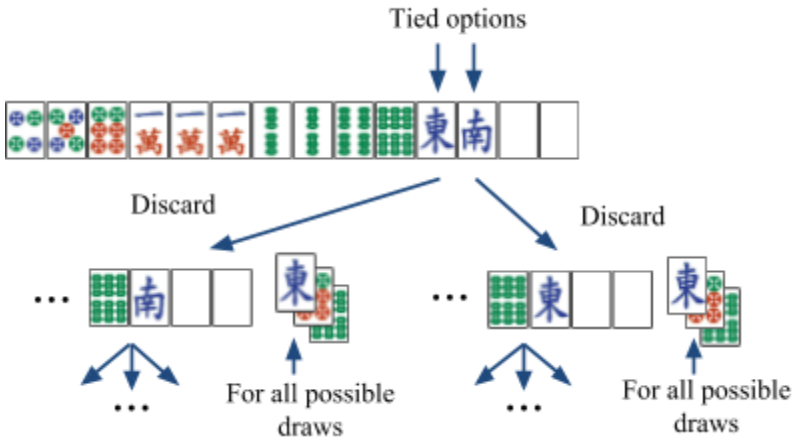
1. Greedy *Shanten* Optimization (Baseline)



As a naive implementation, our most basic AI discards tiles in order to optimize

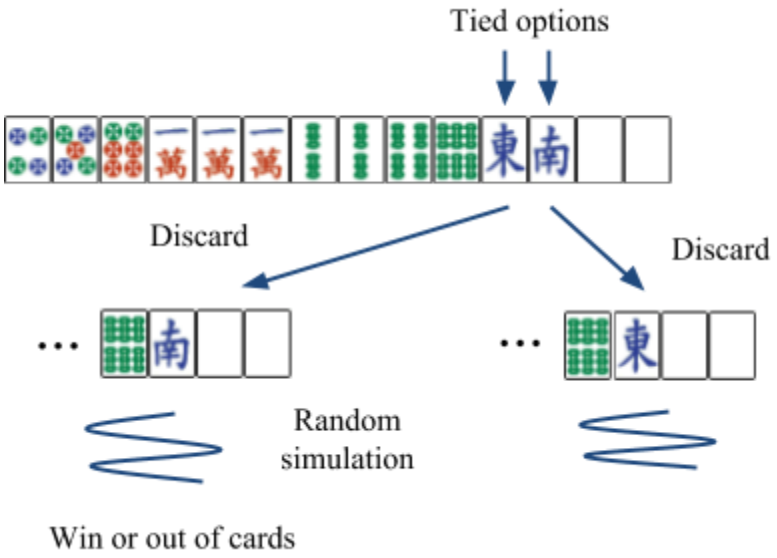
shanten, also only calling melds when *shanten* can be improved. Ties are broken by choosing the first tile found, which is close to random but favors discarding numbered tiles over honor tiles due to the bot's internal implementation. As all other bots use the same initial *shanten*-based elimination but have different tie-breaking methods, we consider this our baseline algorithm.

2. Search-Based *Shanten* Algorithm



The second method is a state-based search, breaking ties by simulating draws and discards for later turns. The resultant search state space is at most (#unrevealed tiles*#discardable tiles)^(*shanten*+1), since a winning hand has a *shanten* of -1; this is reduced by pruning discards that do not improve *shanten*. In other words, the AI discards the tile that lead to the most possible winning hands. This is at worst (34*14)^depth states, and the depth of the search was limited to 3 due to time constraints, effectively considering any sequence of draws each resulting in lowered *shanten* or a win as a successful path.

3. *Shanten* Monte Carlo Search



This method replaces searching all possibilities in the second algorithm with calculating many simulations of a randomly playing agent, where the utility is the negative squared sum of *shanten*. (The lower the sum, the better)

The main challenge of creating a good Monte Carlo simulation scheme is creating an algorithm that can distinguish good hands and bad hands within the 10 second time limit of the platform. Multiple variations of the Monte Carlo simulations were experimented. Since testing the win rate is nearly impossible due to the problems mentioned earlier, our criterion is rather subjective and uses some level of game knowledge. The main method is comparing utility of complete, good (are a few tiles away from winning) and bad hands (hands constructed such that it has minimal pon, chi, or pairs). All algorithms have the the following general algorithm.

Create a vector W representing all possible tiles.

Subtract all tiles in hand, discards, and open melds from W.

For 1,..., Number of iterations:

    For 1, …, Maximum draws:

        Draw a card from W and add it to tiles in the hand

        Calculate the shanten of the hand.

        If hand is complete:

            break

        Discard one card according to the policy

Note that all algorithms do not consider the other players nor calling their discards.

The first scheme counts the number of wins given a player that discards at random. This scheme had a problem that most hands having no wins due to the strict winning rules of mahjong, even with 1000 simulations.

The second scheme weights the discards so that it tries to preserve 3 of a kinds. This method does not check for straits due to the amount of computation needed. This method has similar performance to the first.

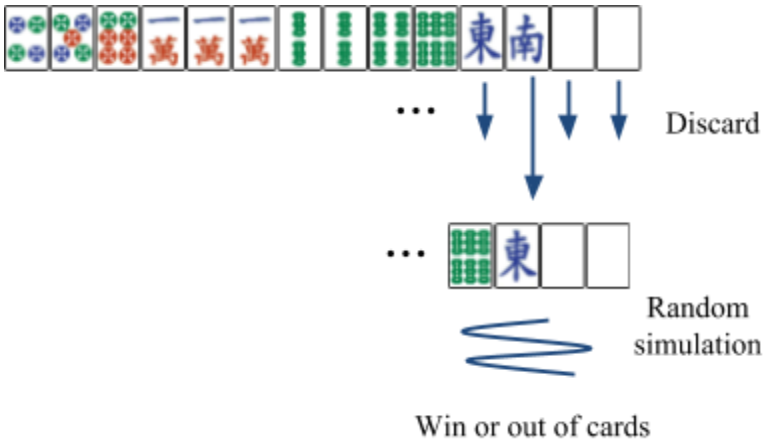
The third scheme was to count the number of *Tenpai* or hands that are won using a randomly playing agent. This scheme however, tends to favor hands that have multiple options rather than complete hands.

The third scheme was to count the negative squared sum of shanten after every draw using a randomly playing agent. The rationale is that hands that can can end quickly, tend to have a low shanten sum, and the square is to penalize hands that take a lot of moves to complete.

The fourth scheme is similar to the third, but replaces the randomly playing agent with Greedy *Shanten* Optimization. This method however is too slow to make sufficient simulations.

Among all these methods, the negative squared sum shanten method at 500 iterations was chosen.

4. Monte Carlo Simulation



This is similar to the Shanten Monte Carlo Search, but does simulations on all discard options instead of tied options.

Results and Analysis

Each AI implementation played several rounds on tenhou.net. The results are summarized in the table below.

	Greedy *Shanten* Optimization (n=678, r=122)	Search -based *Shanten* Optimization (depth = 3, n=899, r=156)	*Shanten* MCS (n=739, r=121)	Monte Carlo Simulation (n=1088, r=193)	Nihisel's AI (provided bot, n= unknown, r=600)
Hand Win Rate (excluding *ryuukyouko*)	0.1077 (0.1189)	0.1835 (0.2010)	0.1407 (0.1583)	0.1489 (0.1667)	0.1997
Feed Rate (excluding *ryuukyouko*)	0.2286 (0.2524)	0.1913 (0.2095)	0.2016 (0.2268)	0.2242 (0.2510)	0.1088
Round Win Rate	0.0328	0.0956	0.0909	0.0933	0.2241
Average Rank	3.37	2.87	3.10	3.03	2.53

Bots were tested for n hands and r rounds, as each round consists of two or more games. Hand win rate is the percentage of hands won, while feed rate is the percentage of hands lost by discarding a tile that caused another player to win. *Ryuukyouko* is the case where a hand ended with no players winning; thus, the statistic for hand win rate excluding *ryuukyouko* and the statistic for feed rate including *ryuukyouko* are more representative of performance. Round win rate is the percentage of rounds in which the AI finished first in point value, and average rank is the mean point value rank amongst the four players. Results from an additional AI provided with the python client are listed (Nihisel's AI); however, the provided bot uses directly coded strategies modeled after professional mahjong logic rather than our strictly state-evaluating AI.

Unsurprisingly, all three complex AI did better than the greedy baseline algorithm, with the search algorithm coming close to breaking the even win rate of 0.25. What was unexpected was that the search-based algorithm did better than both sampling-based algorithms, although the sampling-based ones could explore further depths as a result of not being exhaustive. This is likely since increasing depth looks at increasingly less likely states, with the consistency of an exhaustive search beating out the potential fine-tuning of the sampling.

Variation in feed rate was likely due to optimizations resulting in winning more quickly, as none of our AI were taught how to play defensively and prevent other players from winning. Round win rate and average rank remained unimpressive due to the fact that our AI implementations did not distinguish between different winning hands, usually winning with the lowest possible value.

A few bugs were discovered as we ran our AI. For an unknown reason, the bot could not call *riichi*, or announce a fully hidden hand one tile away from winning, which adds value to a hand and is commonly used as a win condition. In addition, since the bots could not calculate the value of a hand, they would occasionally try to build hands with zero point value when riichi mahjong requires a hand to have value to win, even if the hand is considered complete (a *shanten* of -1). These both suggest that our hand win rate could be pushed even further, probably exceeding that of Nihisel's AI.

Future Work and Conclusions

There are several ways that our AI could be improved upon for future implementations. The current algorithms consider opponent's discards and revealed tiles as one; separating them would allow for analysis of opponent playstyle, better predictions of future tile draws, and defensive play. Obtaining a database of online mahjong games, possibly through reversing our game logs, would allow for us to build learning models to supplement our state analysis and let our AI handle more complex components of the game.

We set out to make a simple AI that could navigate riichi mahjong and produce reasonable win rates. We found success in both simple and complex algorithms, although fine-tuning the more complex algorithms proved difficult given the nature of the game and the restrictions on our processing power. By our evaluation metrics, we near matched existing complex AI, and still have room to improve.

References

[1] Nihisel, Alexey, Mahjong Repository, tenhou-python-bot. GitHub, 2018.

[2] Nihisel, Alexey, Mahjong Repository, mahjong. GitHub, 2018.

[3] European Mahjong Association, Riichi Rules for Japanese Mahjong, 2016.

[4] Loh, Wan Jing, AI Mahjong. CS229, 2009.

[5] Mizukami, Naoki and Tsuruoka, Yoshimasa, Building a Computer Mahjong Player Based on Monte Carlo Simulation and Opponent Models, 2015