Information Set Monte Carlo Tree Search for Rummikub Variant

Austin Chen

Advisors: Professor James Glenn

CPSC 490 Final Report

Fall 2023

Table of Contents

Abstract

Dummykub is a variant of Rummikub, a turn-based game of imperfect information played by 2-4 people. The game state consists of the players' hands, private to each player, the board, the public tiles previously played, and the pool of unknown tiles. There are 2 sets of tiles numbered from 1 to 7 and in three different colors: red, blue, and yellow. Players are initially dealt a hand of 8 random tiles from the pool. Players play sets, which can be either a *run*, a set of three or more consecutive, strictly increasing, numbers all in the same color or a *group*, is a set of three of the same number in uniquely different colors. I will be implementing an IS-MCTS agent. IS-MCTS is an algorithm similar to MCTS, an heuristic algorithm that builds a search tree and expands the tree while focusing on promising branches to find more favorable outcomes; but IS-MCTS uses information sets instead of game states. Information sets are simply sets of states that are indistinguishable from one player's point of view, due to the hidden nature of their opponent's state. I initially started with Monte Carlo CFR (MCCFR) but changed to IS-MCTS after struggling with the large action state. A low amount of games were used and so results varied, but IS-MCTS achieved anywhere from a 70% to an 80% win rate, over a range of exploration values, against a random agent, between 50% to a 60% win rate against a greedy agent. This clearly shows that IS-MCTS performs much better than a random agent, and still better than an impressive greedy agent.

Background

Dummykub is a turn-based game of imperfect information played by 2 people. The game state consists of the players' hands, private to each player, the board, the public tiles previously played, and the pool of unknown tiles. There are 2 sets of tiles numbered from 1 to 7 and in three different colors: red, blue, and yellow. Players are initially dealt a hand of 8 random tiles from the pool. In the starting phase, each player can either draw a tile from the pool or perform an *opening move* by placing tiles on the board in one or more *sets* that total to at least 11 points. There are two types of valid sets:

1. A *run* is a set of three or more consecutive, strictly increasing, numbers all in the same color
2. A *group* is a set of three of the same number in uniquely different colors

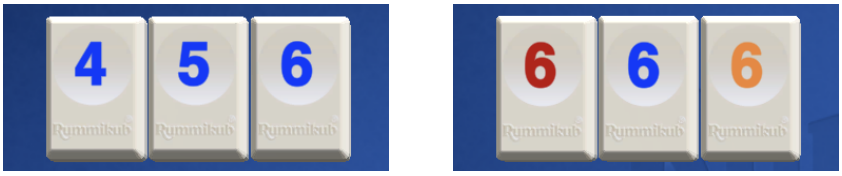

Figure 1: Example of a run and a group

After this opening move, players can draw a tile (if any remain in the pool), place tiles on the board, or rearrange tiles on the board. At the end of a player's turn, the board must be composed of valid groups and/or runs, meaning that a player cannot perform a placement or a rearrangement that leaves the board in an invalid state. The game ends when a player has no more tiles or if there are less than 2 tiles before player 1's turn. The score of each player is calculated by summing up the number of each remaining tile in the player's hand. The objective is to have the smallest total score, played over several games.

Monte Carlo Tree Search (MCTS) is an AI technique that consists of representing game play as a tree, where nodes of a tree represent a specific game state, and edges represent actions leading from one state to another. This algorithm was popularized when programs applying this strategy to Go quickly jumped to "5 dan, which is considered an advanced level" [1]. MCTS operates based on a simple idea, we want to expand the tree based on the most promising move. In order to do so, we need to balance between exploring new moves and exploiting moves that have already been seen to be pretty good. In each iteration of MCTS, we go through 4 steps:

1. Starting from the root state, traverse down the tree through explored actions until you reach a node with no explored actions
2. From this node, choose an action to expand on
3. Simulate the rest of the game from this action, and obtain a result, in our case, the final score for the player to act in the root state
4. Use the result of the third step and backpropagate it upwards to every node traversed in the first step

However, the specifics of how to traverse down the tree in step (1) and how to choose the action to expand on (2) depends on the implementation and may change from game to game.

IS-MCTS embodies the same ideas with one major change, it uses information sets rather than game states [2]. This is because IS-MCTS was developed for imperfect games where a player won't know the true state of the game. In Dummykub, one player doesn't know the tiles in their opponent's hand and in the pool. An information set contains all the possible game states that can't be distinguished from a player's point of view, due to the hidden information.

Implementation

Originally, I planned on applying Monte Carlo CFR (MCCFR) to Dummykub. MCCFR operates on the idea of regret, which is the difference between expected utility of the best action and the utility of the chosen action. MCCFR uses Monte Carlo sampling to update the computed regrets and strategies after each iteration, but the issue is that it needs to be trained on many iterations [3]. Dummykub's action space is incredibly large, since simple rearrangements are considered valid moves. So I expected the computation time for the best move given a game state over many iterations to be quite high. So I looked to IS-MCTS instead.

The first challenge of implementing IS-MCTS was to create an algorithm to find a set of moves. But before I find a set of moves, I need to first create an algorithm to find a valid move. One approach is linear programming where the goal is to maximize the value of the tiles that can be placed onto the board, but also minimize the number of changes of the existing sets on the board [4]. Another approach is variation of a linear time algorithm that solves the Rummikub Puzzle. The Rummikub Puzzle is the following: given the combination of a hand and a board, the goal is to maximize the number of points that can be obtained by making valid runs and groups [5]. The algorithm models this problem by focusing on first constructing runs, then utilizing the leftover tiles to compute groups. Some other ideas include

- Looking for a set of moves but stopping at some depth *d*. However the concept of depth is vague and hard to define
- Assign a value to each tile in a player's hand and board, then focus on removing the *n* most valuable tiles. However, this was still difficult when I moved on to finding a set of moves

In the end, I modified the linear time algorithm that solves the Rummikub Puzzle. Two major modifications I made included separation of required and optional tiles. The Rummikub Puzzle simply takes the union of tiles from the hand and board, and finds the move that yields the largest score. However, it would be an invalid move to not use a tile originally placed on the board. By separating required and optional tiles, and enforcing the constraint of using all required tiles, I can avoid returning these invalid moves. The second modification is to find all legal moves, rather than the max score. This yields a set of moves that I can use, which is required for IS-MCTS.

Results

In order to evaluate my IS-MCTS agent, I played it against a random and greedy agent. However, in Dummykub, oftentimes the most difficult aspect is finding moves, and so a random agent is already quite good, and a greedy algorithm is likely good enough to beat most human players. These graphs present the win rates of the IS-MCTS agent against random and greedy agents, respectively. These win rates are based on 100 games and 100 iterations for the IS-MCTS agent. An extremely difficult aspect of testing this is the low repetition count. 100 games is generally quite low, but each of these simulations took anywhere from 3 to 12 hours, with an outlier at 28 hours.

c	2	2.5	3	3.5	4	4.5	5	5.5	6
Win Rate %	74%	73%	72%	73%	75%	78%	79%	74%	69%

Figure 2: win rates for IS-MCTS vs Random at different exploration constants *c*

Against an agent that finds all valid moves, then picks a random one, IS-MCTS performs at a win rate of anywhere between 70-80%, which is incredibly successful.

c	2	2.5	3	3.5	4	4.5	5	5.5	6
Win Rate %	56%	57%	60%	62%	58%	53%	58%	60%	54%

Figure 3: win rates for IS-MCTS vs Greedy at different exploration constants *c*

Against a greedy agent that finds all valid moves, and picks the one that reduces the score of their hand by the most, IS-MCTS performs 50-60% better, which is still significant. Again, a greedy agent is already incredibly effective because the complexity of Dummykub is often finding moves, rather than picking the best one. One main idea that could result in these results is holding tiles that can be placed regardless of what the board state is. By holding a set, you reduce the possibilities of rearrangements done by your opponent.

Future work

While the implementation of this IS-MCTS agent for Dummykub has shown successful results, there are still many areas that could be improved upon. This implementation of IS-MCTS was significantly restricted by Dummykub's action space. Further work can include additional heuristics to prune recursive function calls and improve move set generation speeds, further work into using depth to find a move set, learning opponents strategies, and expanding Dummykub to 4

players. Eventually, the endgame should be to implement an IS-MCTS agent for Rummikub, where there are more tiles, and more complexities with jokers.

Conclusion

Implementing the IS-MCTS algorithm for Dummykub has clearly yielded great results. MCCFR was initially considered as the algorithm of choice, but was dropped due to the high training time required, and the already huge problem Dummykub's action space presents. IS-MCTS was more effective by abstracting different game states away into information states, and allows the algorithm to handle the hidden aspects of the game, like the opponent's hand. At its core, this project used a linear time algorithm to find a set of valid moves, but there are plenty of other ideas to still be explored. With win rates of 70% to 80% against a random agent, and 50% to 60% against a greedy agent, IS-MCTS clearly outperformed both baseline opponents. However, the low game count per simulation also highlights the difficulty of Dummykub, where finding optimal moves is very time consuming.

References

[1] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. 2012. The grand challenge of computer Go: Monte Carlo tree search and extensions. Commun. ACM 55, 3 (March 2012), 106–113. https://doi.org/10.1145/2093548.2093574

[2] P. I. Cowling, E. J. Powley and D. Whitehouse, "Information Set Monte Carlo Tree Search," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 4, no. 2, pp. 120-143, June 2012, doi: 10.1109/TCIAIG.2012.2200894.

[3] Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. 2009. Monte Carlo sampling for regret minimization in extensive games. In Proceedings of the 22nd International Conference on Neural Information Processing Systems (NIPS'09). Curran Associates Inc., Red Hook, NY, USA, 1078–1086.

[4] D. Den Hertog and P. B. Hulshof, "Solving Rummikub Problems by Integer Linear Programming," in The Computer Journal, vol. 49, no. 6, pp. 665-669, Nov. 2006, doi: 10.1093/comjnl/bxl033.

[5] Rijn, Jan N. van et al. “The Complexity of Rummikub Problems.” ArXiv abs/1604.07553 (2016): n. pag.