Scripts of Tribute AI Competition - SOISMCTS Bot

July 16, 2024

1 Introduction

This document discusses the algorithm and test results for an Information Set Monte Carlo Tree Search (ISMCTS) bot developed for the Tales of Tribute card game.

2 Description of the algorithm

There are a number of different components to the algorithm, that are described below.

2.1 Information Set Monte Carlo Tree Search

At a high level the algorithm follows the single observer ISMCTS algorithm proposed in Cowling et al. (2012). Pseudo code for this algorithm can be found in appendix A of this paper under the SO-ISMCTS approach. The main difference here is that the equivalence relation used to define information sets (IS) is based on the move history from the root to that IS node, as opposed to an equivalence relation defined with respect to hidden information. Additionally, we only create and explore a tree corresponding to the current player's turn, no simulation into the enemy player's turn occurs. This is because the average number of moves per turn, can be around 10 and simulating into the player's turn is computationally demanding.

2.2 Node re-use

At the end of each move, the IS node that results from the best chosen move is re-used as a starting point for building the IS tree for the subsequent move. The sub-tree defined as all nodes reachable from the chosen node is maintained, along with the relevant individual node statistics, such as visit count, availability count and total reward.

2.3 MAST

The moving average sampling technique (MAST) is used to adjust moves selected during rollout. This is done by firstly tracking for each move the cumulative total reward and the number of times it has been visited during the rollout phase. These MAST statistics then re-feed back into move selection during the rollout phase by selecting a move, x, from a set of possible moves M, based on a probability given by,

$$Prob(x) = \frac{e^{\frac{TR(x)}{C(x)T}}}{\sum_{y \in M} e^{\frac{TR(y)}{C(y)T}}}, \tag{1}$$

where $TR(x)$ is the total cumulative reward across playouts for move x, and simikalrly $C(x)$ is the number of times the move x is chosen. Finally, T is a temperature parameter. Note we also use a parameter ϵ, to

randomly choose between selecting the next move in our rollout based on the above Gibbs distribution versus just selecting a move at random.

2.4 Computational time allocation

The time limit to select a move in a given turn is computed by taking the time limit for a turn (10s according to competition rules) and dividing by the expected number of moves for that turn. The expected number of moves per turn tends to start off at around 8 or 9 at the start of a turn, increase to around 10 moves in the middle of a turn and then trail off down to one or two moves when the turn is close to ending. To calibrate this we ran 100 games against last year's competition winner and calculated the expected number of moves per turn using the dataset of 100 games.

2.5 Elements re-used from previous year's winner

Finally in addition to the above, the algorithm filters moves using the same approach as last year's winner. So that is it quickly plays 'obvious moves' and reduces the tree size. It also reuses the heuristic designed to evaluate game state from last year's winner (and also optionally the heuristic used by the MCTSBot).

3 Structure of the code

3.1 Main loop

The main monte carlo tree search algorithm is executed within a while loop inside the Play function. The main steps in this loops are as follows,

1. A new determinisation is provided to the root node of the tree. A determination is an object that encapsulates a seeded game state (i.e. state of the game where all randomness has been removed), and a set of possible moves for the current player from that state. During each iteration we only explore those actions compatible with the determinisation set at the root node.

2. The next step is to select a node from the tree for expansion (select function). This uses an upper confidence bound (UCB) multi-armed bandit algorithm. At each node, if it has any children in the tree compatible with the current determinisation, we chose the child with the highest UCB score and move to the next node in the tree. This continues until we reach a node where there are no compatible children in the tree or we reach a game end state or we reach an end turn move. This is then our selected node. Note that we don't select nodes corresponding to an enemy player's turn as we only simulate for the current player.

3. Next we expand our selected node (expand function). We do this by choosing at random (or using MAST statistics) a move from the list of moves from our selected node that are compatible with the current determinisation, and also do not yet have a child node in the tree. We then create a new node as a child node of the selected node with a determinisation given by applying the chosen move to the seeded game state in the selected node. Note, if the only move compatible with current determination from the selected node is an end turn move, then we do not expand, as we consider only the tree relevant for the current player?s turn.

4. After we have our expanded node, we preform the rollout simulation (rollout function). This calculates an expected reward for the current player playing from the expanded node, by simulating moves (either through random choice or based on MAST statistics) until we are left with only an end turn move.

5. Finally we have a backpropagation step (backpropagation function), where each node that was stepped through using the multi-armed bandit algorithm in the selection step and the expanded node, has it's visit count, availability count, max reward and total reward updated. The availability count determines the number of times it could have been selected when stepping through the selection procedure.

After our MCTS loop is completed (based on computational time allocated), we then choose the best move from the root node (chooseBestMove), by selecting the child node with the highest max reward value. This move is then returned back to the game engine by the play method.

3.2 InfosetNodes

InfosetNode is the object type used to define the information sets in our monte carlo tree. Each node contains a number of key data attributes. Firstly, we have the usual parent and children nodes, and statistics such as total and max rewards, visit counts, etc. Next we have the reference move history (_refMoveHistory). This contains the list of moves from root to this node, when the node was first generated. This provides a reference state for the equivalence relation between information sets.

Next we have the current determinisation, which contains the seeded game state and move list attached to this node for the current loop in the MCTS. We also have the current move history, which is the list of moves used in the current MC loop to reach this node (this should be the same as the reference move history, so is slightly redundant but also useful for error checking). Next we have the list of moves based on the current determinisation from this node that have no children in the tree (also labelled uvd in the code), and secondly the list of children in the tree which are compatible with the current determinisation (also labelled cvd in the code). Both uvd and cvd are computed when needed in the function calcChildrenInTreeAndMovesNotInTree. They are both key quantities used in the MC loop described in the previous section.

Next we have the checkEquivalence function that checks if a move history is the same as the reference move history for a node, and hence we have two equivalent information sets. This uses the MoveComparer object which provides a hash code for each move (which are also combined to provide a hash key for the information set). This hash code does not include the unique ID for a card, so that two identical cards give equivalent moves.

As a final remark, the SetCurrentDeterminisatioAndMoveHistory function is used to update the determinisation associated with the root node during each iteration of the monte carlo loop. It also subsequently clears the determinisation from all child nodes. The current determinisation is then only used again when we are expanding and creating a new node. We also reset the current determinisation of children in the tree which are found to be compatible with the current determinisation when we call calcChildrenInTreeAndMovesNotInTree. This ensures that as we move through a tree and find compatible children in the tree (based on the IS equivalence relation), that they also contain the current determinisation for this loop through the simulation.

3.3 Run configuration

The bot can be configured using the run configuration parameters at the start of the class definition. The treeReuse boolean allows for the toggling on and off of tree reuse between moves. The randomDeterminisations allow for switching on and off the random determinisations, the random determinisations are sampled with a frequency given by noSimsPerRandomisations. If this is set to say 100, only once every 100 iterations of our main MCTS while loop will the random determinisation be reset at the root node.

MAST can be toggled on or off using useMAST, and the temperature parameter in the Gibbs distribution function can be changed using the temperature member variable. The thresholdForRandomMoves should be set between zero and one, and gives the probability of randomly choosing each move during

the rollout phase as opposed to using MAST. We can also toggle on and off move filtering using the filterMoves boolean. Move filtering is reused from last year's winner.

If the useMCTSBotHeuristic is set to true, our ISMCTS bot uses the same heuristic as the MCTSBot, if it is set to false we use the same heuristic as the BestMCTS3 bot. Next we have the K value which weights the exploration term in the UCB calculation used for node selection.

Next we have parameters that determine the time allocation. If timeAllocation is set to true, the time per move is given by the time limit per turn divided by the expected number of moves in a turn. The latter is hard-coded in the dictionary expNoMovesPerTurn. If timeAllocation is set to false, then the timeForMoveComputation is set to a fixed value in the PrepareForGame function.

The remaining member variables cover various counters that are used for reporting (generated in the gameEnd function), setting up the logs and initialising random number generators.

4 Test results

Here are a selection of test results using different configurations of the bot.

Random Determinisations	Node Reuse	MAST	Budget allocation by move	Flat Budget Alloc	Move Filtering	Best move criterion	Heuristic	No of Games	Win Rate (vs.BestMCTS3)	Win Rate (vs.MCTSBot)
Y	Y	N	N	0.3	Y	MaxReward	BestMCTS3	50	16%	
Y	N	N	N	0.3	Y	MaxReward	BestMCTS3	50	28%	
N	Y	N	N	0.3	Y	MaxReward	BestMCTS3	50	50%	
N	N	N	N	0.3	Y	MaxReward	BestMCTS3	50	64%	
N	N	N	N	0.3	Y	MaxReward	BestMCTS3	500	48.6%	
N	N	N	N	0.3	Y	MaxReward	BestMCTS3	500	47.8%	
N	Y	N	N	0.3	Y	MaxReward	BestMCTS3	500	44.4%	
N	Y	N	N	0.3	Y	MaxReward	BestMCTS3	100	34%	
N	Y	N	N	0.3	Y	MaxReward	BestMCTS3	100	42%	
Y	N	N	N	0.3	Y	MaxReward	BestMCTS3	500	27%	
Y	N	N	N	0.3	Y	MaxReward	BestMCTS3	500	30.8%	
N	N	Y	N	0.3	Y	MaxReward	BestMCTS3	500	46%	
N	Y	Y	N	0.3	Y	MaxReward	BestMCTS3	500	40.6%	
Y	N	N	N	0.3	N	MaxReward	BestMCTS3	500	20.4%	
Y	Y	Y	N	0.3	Y	MaxReward	BestMCTS3	500	22.8%	
N	N	N	N	0.55	Y	MaxReward	BestMCTS3	500	47%	
N	N	N	Y	N/A	Y	MaxReward	BestMCTS3	500	39.8%	
Y (modulo 100)	N	N	N	0.55	Y	MaxReward	BestMCTS3	500	40.2%	
N	N	N	N	0.3	N	MaxReward	MCTSBot	100		54%
N	Y	N	N	0.3	N	MaxReward	MCTSBot	100		52%
N	Y	N	N	0.55	N	MaxReward	MCTSBot	100		54%
N	N	N	N	0.3	N	VisitCount	MCTSBot	100		0%
N	N	N	N	0.55	N	VisitCount	MCTSBot	100		0%
N	N	N	N	2.5	N	VisitCount	MCTSBot	25		0%
N	N	N	N	10000 iterations	N	VisitCount	MCTSBot	100		3%

5 Outstanding questions

1. How could we take trashing cards more effectively into account? Since we don't simulate into future turns where the benefit of trashing becomes apparent. It is unclear as to how the current heuristics capture this.

2. Why is the best move selection criteria of node visit count performing so poorly, even with a large number of iteration per move?

3. Why does turning on random determinisations, node reuse and/or MAST generally lead to poorer perfomance versus the 48.6% win rate achieved with having them all turned off?

4. Why does increasing the time allocation per move from 0.3 to 0.55 lead to a slightly reduced win rate?

5. Is max reward a good selection criteria as the move choice tends to be based on quite noisy statistics? is this why the other bots tend to use max reward as opposed to visit count?

6. Is it possible that having more simulations centred on a single sample from our root IS (i.e. having random determinisations turned off) leads to better results than when we have more random determinisations as these introduce more MC noise into the node stats?

7. Is the lack of improvement from node re-use and MAST due to the selection of move based on max reward requiring much less simulations and hence these improvements dont have a big impact?

References

Cowling, P., Powley, J. and Whitehouse, D., 2012. Information Set Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4 (2), pp. 120-143.