Exercise 1
Ultimate Tic-Tac-Toe
Artificial Intelligence for Games

Hubert Obrzut

October 21, 2020

1 Base Algorithm

- Implemented algorithms: **Flat MC** and **MCTS** with UCT selection policy and random simulation policy as specified in the contents of the exercise.
- Programming language used: **C++**.
- Testing: performed both on my own machine - making agents play against each other - and CodingGame platform.
- Number of playouts gathered (on my own machine): **1000**.
- Total number of seconds running: **4534.2s**.
- Time limit for one move: **100ms**.
- Exploration constant for UCT policy in MCTS: **0.4**.

1.1 Algorithm results

We define simulation as a one iteration of the algorithm - for example one sequence of selection-expansion-simulation-backpropagation is the one simulation in MCTS algorithm.

	FlatMC	MCTS
Winrate against each other	2.6%	89.4%
Average number of simulations per turn	658	586
Coding Game Top Players	53%	34%

Testing on Coding Game platform was done as a Gold Ranked player - numbers of players in total, when testing was done - 596. Although we can say that pure MCTS dominates FlatMC on my own machine, in the Coding Game environment the difference is not so huge.

Note: Winrates do not add up to 100% because, of course, we can have a draw.

2 Enhancements

I have decided to implement two enhancements from the list in the exercise: **MAST** and **RAVE**. I though they would improve effectiveness noticeably, but the major improvement has not been seen. To make sure that I did not make any mistakes in the algorithms, I have rewritten them a few times and analyzed carefully. Results of these algorithms and comparisons between them are presented below.

3 Parameter Tuning

In order to tune the parameters, firstly, I have decided to compare them to the previous agents I have written. FlatMC agent was the first one (actually pure random agent was the first one, but it was mainly used for testing the logic of the game, and whole design of the program) - it has fairly straightforward implementation and no parameters to tune, so it was pretty good to use. Also it turned out that with FlatMC policy you can get to the Gold League, so the agent is not so "stupid". As I was writing the next agent, I tried to get the best winrate against FlatMC. The advantage of that approach is that it is fast - especially because I have abstracted specific parts of the algorithms, so the battle between two agents consisted of one change in the code. This gave me the general idea of what are the best configurations of the parameters for a particular agent. As I had more agents I had more possibilities to test the parameters (so later on I also tested, e.g. MAST with pure MCTS, and so on ...). Also, not to rely only on my implementations, I have used Coding Game platform at first, and then CGBenchmark (against three opponents) in order to check if my configurations deal with other people agents. I had to tune the following parameters:

- `exploreSpeed` - exploration constant in the UCT selection policy, tested values: $[0.1, 1)$ with $step = 0.1$
- `epsilon` - ϵ constant in ϵ-greedy simulation policy in MAST enhancement, tested values: $[0.1, 1)$ with $step = 0.1$
- `decayFactor` - γ factor used to decay evaluations of actions in MAST policy from the previous rounds, tested values: $[0.1, 1)$ with $step = 0.1$
- `K` - factor in AMAF state-action evaluation used in RAVE policy, tested values: $[5, 10, 20, 50, 100, 200, 500, 1000]$

With the methods described above the optimal value for all the parameters are as follows:

`exploreSpeed`	**0.4**
`epsilon`	**0.8**
`decayFactor`	**0.6**
`K`	**50**

4 Method Comparison

I have tested algorithms against each other on my own machine. Every comparison was made by taking the average score of 1000 playouts between the two particular agents. All of the parameters of the agents were set to be optimal.

Agent1 \ Agent2	FlatMC	MCTS	MAST	RAVE
MCTS	2.6% / 89.4%			
MAST	6.1% / 88.4%	34.2% / 39.7%		
RAVE	2.4% / 89.7%	31.4% / 35.2%	30.3% / 37.4%	
MAST with RAVE	2.9% / 89.1%	33.4% / 39.3%	36.6% / 38.8%	33.4% / 33.9%

Table 1: Algorithm comparison - winrates

In addtion I have tested all of the variants in the Coding Game environment. Here are the results:

Agent	Coding Game Top Players
FlatMC	53%
MCTS	34%
MAST	37%
RAVE	36%
MAST with RAVE	37%

Table 2: Algorithm comparison - place in the Coding Game platform in Gold League

We can see that we obtain better results with MAST or/and RAVE againt MCTS. The difference is not huge - we can also notice that a lot of times, when playing against pure MCTS, ends with draw, so we can suppose that variants play pretty equally - with one being a little better that the other. However one peculiar thing is that when it comes to the CodingGame platform, the pure MCTS deals with the opponents the best.

There can be a few reasons why these enhancements do not improve much and when it comes to the CG environment, they even worsen the score. First of all 100ms for one turn could be too little time, in order for these techniques to gather enough data, so that it is meaningful - we know that for example in RAVE, we keep local evaluation of all the actions played down in the Monte-Carlo tree - these evaluations are based on the premise that one move is good independent from all of the other moves. So for this statistics to be reliable we have to gather some data, because they are already in some way disconnected from the current game state.

Another reason could be that MAST and RAVE techniques are not the best choice for Ultimate Tic-Tac-Toe game. Maybe the dependency between the state and the action is too high to use state-independent action evaluations.

The last thing is that, of course, the more time we give for one turn, the better the agent is playing. If MAST and RAVE enhancements are not the best choice for this game, if you use them, you also waste the potential of more calculations, if you have used other techniques. As we know MAST and RAVE require a little more memory and time consumption so that potential waste, could also be the reason why they do not improve too much.