Study and Optimization of Monte-Carlo Tree Search for Backgammon

Clara Gard, Liora Taieb

April 2024

1 Introduction

Backgammon is a board game in which each player rolls dice then determine their next move. Its inherent randomness makes it an interesting problem to explore with MCTS. But because of it, a beginner can sometimes beat an expert. The game has a quite large number of possible moves, resulting in non-negligible computational complexity.

We have compared the performances of random (non MC), FlatMC, UCT, GRAVE and PPAF, putting them up against each other. As described below, given the important amount of time required to run those algorithms, we had to reduce the number of playouts and games, making the results less significant but still insightful.

2 Preliminary study

We have based our Backgammon implementation on the code found in the following GitHub page:
https://github.com/weekend37/Backgammon/blob/master/Backgammon.py.
We optimized it to remove clear redundant moves in the original finding of legal moves. Our code is about 2.5 times faster and suppress around 40% of unnecessary moves. It is stored in the file *"Backgammon.py"*. The file *"Backgammon2.py"* is an attempt to partially code the ideas of optimization in [1].

First and foremost, we have observed the results of 10 000 games with 2 random players, (also see *"Backgammon.py"*. We tested the hypothesis that starting yields 50% of victory. The confidence interval with p-value 5% and that many games is [49%, 51%]. We observed the beginning player winning mostly between 50% and 51% of games (we re-ran the experiment several times), refuting a significant advantage in starting. Let us note that reiterating the experiment, this time allowing doubles as first dice roll (forbidden in official Backgammon rules), yields a beginning player winning mostly above 51.5% of times. Doubles as first move would secure a slight but significant advantage in starting, the game is pretty fair.

Running 1000 random games took us 12s to compute. Each game lasts a mean of 45 moves per player, computing 100 games of FlatMC with 1000 playouts against random, each turn of FlatMC exploring a mean of 30 legal moves, would take us 18.75 days. We chose to run code that maximize the number of games and playouts while computing in 1 to 3h, hoping still to extract interesting behaviour. This also limited our desire to study the first move as done in [1], that we ended up omitting.

3 Monte Carlo algorithms

We show next our results of the confrontation between random, FlatMC, UCT, GRAVE and PPAF. The code for each method can be found in the associated .py file. Let us note that we have also coded RAVE and Sequential Halving, but both were way too slow to be used.

Conditions		Opponents		Winning rate	Dice rolls per player
Games	Playouts	Opp. 1	Opp. 2	Opp. 1 over Opp. 2	Average per game
10	50	FlatMC	Random	80 %	48
50	100	UCT	Random	96 %	39
10	50	UCT	FlatMC	90 %	38
50	100	GRAVE	Random	84 %	40
20	50	GRAVE	FlatMC	60 %	38
20	500	GRAVE	UCT	5 %	28
10	50	PPAF	Random	80 %	42
15	50	PPAF	FlatMC	87 %	48
20	50	PPAF	UCT	20 %	43
20	50	PPAF	GRAVE	55 %	38

Table 1: Winning rate of MC algorithms over each other in Backgammon.

Unsurprisingly, every algorithm is largely better than random, and every algorithm except random is better than FlatMC. UCT is better than anybody else, and appears as the best Monte-Carlo choice among the studied algorithms for Backgammon. We can note that UCT, as well as being the best, is the fastest winner, followed by GRAVE. GRAVE, although quite fast, is not as good as expected. We believe it is due to the big uncertainty of dice roll during the game. In Backgammon, learning from the past playouts could be more confusing that anything else. PPAF seems like the second best choice among our tested algorithm. Unlike GRAVE, PPAF doesn't learn from the past, but it does try to construct a policy in an uncertain environment. It should be studied in itself, and allocated (a lot) more resources to do so.

4 Conclusion

Our project mainly showed one thing: the importance of very well optimized game code on which applying Monte-Carlo algorithms. We could have dug into the parallelization of games, if not of exploration inside one game. To further develop getting more interesting results in less time, it could have been interesting to explore pruning policies and/or better tailored algorithms that include features/bias on moves, such as eating or being potentially eaten.

References

[1] F. V. Lishout, Guillaume Chaslot, J. Uiterwijk, *Monte-Carlo Tree Search in Backgammon* (2007), https://orbi.uliege.be/bitstream/2268/28469/1/vanlishout_backgammon.pdf