Monte Carlo 6 Nimmt!

Yanis Bakhtaoui and Tristan Cazenave

LAMSADE, Université Paris-Dauphine, PSL, CNRS, Paris, France
Tristan.Cazenave@dauphine.psl.eu

Abstract. Monte Carlo Tree Search (MCTS) algorithms have recently been used to solve complex games. The most well known examples are the games of Chess and Go, in which the best human players can't compete with some algorithms. We propose to implement different variations of these algorithms on the game of ”6 Nimmt!”. The variations used are Upper Confidence for Trees (UCT), Playout Policy Adapation (PPA) and Most-Average Sampling Technique (MAST). The experiments compare 4 variations between them : a naive approach, UCT, PPA and MAST.

1 Introduction

Monte Carlo Tree Search is useful in complete information games [1]. We are interested in the card game ”6 Nimmt!”. It is a simultaneous moves game.

In this study, we will focus on the use of some MCTS algorithms to solve the two player version of ”6 Nimmt!”. Although the original version of ”6 Nimmt” is a multi-player game, the pro version is only played by two player, so that each player knows the card of his opponent. They can therefore develop a more complex strategy. We will implement playout policies with move features [2]
Contrary to the games of Go and Chess, ”6 Nimmt!” doesn't have an infinity of possibilities. We remind that these two games have respectively 2 x 10^{172} and 10^{120} game possibilities.
For our game, if we consider that the cards are already distributed, we have $(10!)^2 \sim 10^{13}$. The total number of initial distributions is $\sim 10^{10}$. So, if we also take in consideration the different hand cards distributions, the number of possible games goes up to 10^{23}, which is a lot less than Chess. However, it is still a high number of possibilities, so the MCTS algorithms are applicable.
During this study, we will consider that the game is already set, which means that each player has 10 cards randomly distributed, and the board has 4 cards. The 10^{13} possibilities then corresponds to the order in which the players play their cards.
It is important to notice that, for a few configurations of the game, the game cannot be won. Thus, to make the randomness of the starting hand insignificant, we will run 500 games for each algorithm tested.

2 Monte Carlo Search Algorithms

We will first present Flat Monte Carlo and UCT with Simultaneous Moves. To play the game of ”6 Nimmt!”, the classical version of MCTS using UCT is really effective, and pretty fast. Some variations of the algorithm exist [4,2]. Although the body of the algorithm (explore, expand, simulate, backpropragate) remains the same, we can change the way we perform playouts.

2.1 Flat Monte Carlo

A naive first approach would be to simulate a high number of games for each playable card, and then play the card with the higher number of success. This approach relies on Monte Carlo estimators, and if we simulate a high enough number of games, the victory rate will converge to the expectation of winrate. Then, playing the card which has the higher expectation of victory will increase the odds of winning the game.

```
FlatMonteCarlo(nbr_simu, actualGame, handPlayer1)
for move in possibleMoves do
    move.wins ← 0;
    for n in 1, ..., nbr_simu do
        copy ← actualGame;
        play(copy,handPlayer1[move]);
        gameResult ← RandomSimu(copy);
        if gameResult = true then
            move.wins++;
        end
    end
end
return The move with the most victories;
```

Algorithm 1: Flat Monte Carlo

Using this algorithm, we get good results against a player who plays randomly.

Number of simulations in the algorithm	100	200	500	1000
Winrate	96,7%	96,8%	96,9%	97,5%
Time (in microseconds)	689258	1434736	3919425	7055444

Fig. 1: Performances of the Monte Carlo Flat algorithm - playing 500 games against a player playing randomly

The more simulations we generate at each step (a step being the choice of the card to play) of the algorithm, the more complex the final algorithm becomes. Indeed, if we add 1 simulation at each Monte Carlo Flat simulation, then for the 1st step, we will have 10 more simulations, for the 2nd we'll make 9 more, and so on. So, if we add 1 simulation, we'll end up adding a total of $\sum_{i=1}^{10} i = \frac{10 \text{ x } 11}{2} = 55$ simulations. When we add 300 simulations, we add 300 x 55 $=$ 16500 simulations, which explains the difference of cost.

2.2 UCT with Simultaneous Moves

In the Flat Monte Carlo algorithm, we simulate an equal number of games for each possible move. It works, but it costs a lot. Some moves are bad, and if we could detect which won't lead to success, we could prevent simulating a bad move. By doing so, we would limitate the number of simulations, and so the complexity of the algorithm.
One way to do that is to use the MCTS algorithm. We consider each playable move for the player as nodes. The root node has 10 subtrees (the 10 possible moves of the player using MCTS). Then each one of these 10 nodes has 10 subtrees (the 1O possible moves of his opponent). They all have 9 subtrees (the 9 remaining cards of the first player), and so on. The algorithm will simulate games to identify the good moves, just as for our Flat Monte Carlo algorithm. But, instead of simulating equally all moves, we will determine which move to simulate according to the result of the previous simulations.
As in "6 Nimmt!" players play simultaneously, we will consider at each step 2 nodes : the one of the player using MCTS, and the other one corresponding to the card that the other player will simultaneously play. By doing this, the algorithm will take into account that both players play simultaneously, and won't play the first card then the second one.

```
MCTS(nbr_simu, actualGame);
for i in 1, ..., nbr_simu do
    road ← UCT(actualGame);
    copy ← road;
    winner ← RandomSimu(copy);
    adapt(road, winner);
end
return The move with the most playouts
```

Algorithm 2: Monte Carlo Tree Search

This algorithm should be faster than the previous one, as it performs less simulations. In addition, it could be more precise, as it doesn't completely randomly simulates the playouts, but does it only from the leaf of the tree.

The constant c in UCT, named "exploration parameter", determines the balance between exploration and exploitation. We will optimise it for our game.

c	0,2	0,3	0,5	0,7	0,9	1,1	1,3	1,5	1,7	**1,8**	1,9
Winrate	91%	94,2%	95,8%	95%	96,8%	96%	96%	96,4%	96,2%	**97,2%**	96%

Fig. 2: Performances of the Monte Carlo Tree Search algorithm, depending on the exploration parameter

```
UCT(actual, c)
while actual.sons $\neq$ NULL do
    bestValue $\leftarrow -\infty$
    for move in possiblesMoves (actual) do
        t $\leftarrow$ axtual.games [move]
        w $\leftarrow$ actual.wins [move]
        total $\leftarrow$ actual.games
        value $\leftarrow \frac{w}{t} + c$ x $\sqrt{\frac{\log(total)}{t}}$
        if value $>$ bestValue then
            bestValue $\leftarrow$ value
            bestMove $\leftarrow$ move
        end
    end
    actual $\leftarrow$ play (actual, bestMove)
    road.add(bestMove)
end
return road
```

Algorithm 3: UCT

```
adapt(road, winner);
for move in road do
    move.games++;
    if winner = move.player then
        move.wins++;
    end
end
```

Algorithm 4: Adapt

Number of simulations in the algorithm	100	200	500	1000	1500
Winrate	93%	93,5%	96,2%	**97,2%**	96,5%
Time (in microseconds)	116251	217963	561017	1121831	1581753

Fig. 3: Performances of the Monte Carlo Tree Search algorithm, with 500 games played

As expected, the running time of the MCTS algorithm is much lower than the MC flat's one - for similar winrates. The MC flat with 200 simulations in each stage completes in approximately 1.434.736 ms and gets a winrate of 96,8% against a player playing randomly. The MCTS with 1000 simulations, on the other hand, completes in 1.121.831 ms and wins 97,2% of its games.

Even though it is interesting comparing the performances against a random player, we can compare UCT and Flat playing each other.
To compare the different algorithms, we will simulate games. Playing games with one agent playing with one algorithm and the other one with the other algorithm would not be precise, as the outcome depends on the original distribution. We use duplicate scoring to counter this [3]. The different algorithms will play against the same agent, with the

same initial board. We will only retain the parties for which one of the algorithms defeat the defender while the other loses. We will take UCT with 1000 simulations as defender for all simulations.

Algorithm used	MC flat 200	MC Flat 500	MC flat 1000
MCTS 500	**54,3%**	49,7%	46,9%
MCTS 1000	**66%**	**59,3%**	**50,3%**
MCTS 1500	**66%**	**59,2%**	**50,5%**

Fig. 4: Comparison of MCTS algorithm and Flat Monte Carlo playing against the same agent, for 300 games that have different results.

From a certain number of simulations, both algorithms perform in the same way (all results are close to 50%). So the only real difference between both algorithms is the running time.

2.3 Playout Policy Adaptation

The first variation proposed is Playout Policy Adaptation (PPA). As in the classical MCTS algorithm, we explore the tree using UCT. The difference is on the way we simulate the playout when reaching a leaf on UCT. While in the classical algorithm we simulate randomly, in PPA we will control the playout with weights for each node. At each step of the simulation, we choose the next node to explore randomly. The choice is randomly made between possible moves proportionally to the weights.
The associated weights are then adapted according to the final result of the playout.
First, we will only consider in the playout the card played. Then, we won't only consider the moves as the associated card played, but we will also consider the step in which the card is played. By doing so, we are able to determine if a move is interesting at a specific time of the game, and not only if it's a good move in overall. We will note "PPA without features" the algorithm that considers only the card played, and "PPA with features" the one taking into consideration the step in which the card is played.

```
playout_algorithm(state, policy, k)
while not terminal (state) do
    z ← 0.0
    for move in possibleMoves (state) do
        z ← $z + exp(k * policy[move])$
    end
    Choose a move for player with probability proportional to $\frac{exp(k*policy[move])}{z}$
    road_playout.add(move_chosen)
    state ← play (state, move_chosen)
end
return winner, road_playout
```

Algorithm 5: Playout algorithm

```
adapt_PPA(winner, road_playout, policy, $\alpha$)
polp ← policy
for move in road_playout do
    if winner = move.player then
        polp [move] ← polp [move] + $\alpha$
        z ← 0.0
        for son in move.sons do
            z ← $z + exp(policy[son])$
        end
        for son in move.sons do
            polp [son] ← $polp[son] - \alpha * \frac{exp(policy[son])}{z}$
        end
    end
end
policy ← polp
```

Algorithm 6: Adapt PPA

```
PPA(actualGame, $\alpha$, policy, c )
for i in 1, ..., nbr_simu do
    road ← UCT(actualGame, c)
    copy.simulate(road)
    winner, road_playout ← playout_algorithm(actualGame,policy,k,road)
    adapt_PPA(winner, road_playout, policy, $\alpha$)
end
return The move with the most playouts
```

Algorithm 7: PPA

The results of the PPA algorithm are similar to the MCTS's ones. But, as we need to adapt the policy at each playout, it is more complex.
We tried several values for the α constant, and the best one seems to be 0,4.

2.4 Move-Average Sampling Technique

The second algorithm proposed is MAST [4]. It differs from PPA in its way of assigning weights, and the way we code moves.

Alpha	0,1	0,2	0,3	**0,4**	0,5	0,6	0,7	0,8	0,9
winrate PPA without features	95,4%	95%	96,1%	**97,3%**	95,8%	96,4%	95%	92,6%	90,4%
winrate PPA with features	96,2%	96,1%	96,8%	**97,5%**	95,8%	96,3%	94,1%	91,5%	92,4%

Fig. 5: Winrates of the PPA algorithm with and without features, depending on the alpha constant - playing 500 games against a player playing randomly

```
adapt_MAST(winner, road_playout, policy);
for move in road_playout do
    games[code(move)]++;
    if winner = move.player then
        wins[code(move)]++;
    end
    move.policy ← $\frac{wins[code(move)]}{games[code(move)]}$
end
```

Algorithm 8: Adapt MAST

k	1	1,5	2	2,5	3	**3,5**	4	4,5	5
winrate MAST without features	94,5%	94,3%	94,4%	95,9%	96%	**97,7%**	96,3%	94,8%	94,7%
winrate MAST with features	95%	95,5%	94,5%	96,2%	97,3%	**98,3%**	96,4%	95%	96,1%

Fig. 6: Winrates of the MAST algorithm, depending on the k constant - playing 500 games against a player playing randomly

As for the PPA algorithm, we tried several values for k, and the best seems to be 3,5.

For the simulations of the PPA and the MAST algorithms, we used 1000 simulations, as it was the most efficient for the classical MCTS algorithm. The respective execution times are 1.617.286 and 1.487.276 ms. This small difference is explained by the fact that for the MAST algorithm, we only update the policies of the nodes used, while for the PPA we also update the policy of the nodes' sons.
To compare the versions with or without features, we used duplicate bridge again. PPA taking the step in which we play the card won 55,7% of its games against PPA with only the number of the card. MAST with features won 57,1% of its games against naive MAST.
Finally, both of these algorithms give similar results than the classical MCTS algorithm,

```
MAST(actualGame, k, policy, c );
for move in possibleMoves do
    wins[code(move)] ← 0;
    games[code(move)] ← 0;
end
for i in 1, ..., nbr_simu do
    actual ← actualGame;
    road ← UCT(actual, c);
    winner, road_playout ← playout_algorithm(actual,policy,k);
    adapt_MAST(winner, road_playout, policy);
end
return The move with the most playouts
```

Algorithm 9: MAST

but are more complex. Indeed, the method of simulation, and the update of policies cost a lot of computational time.

3 Experimental Results

To compare the different algorithms, we will use again the duplicate bridge method. As the MAST gave the best results playing against a random player, we will use MAST as the defense.

A move can be associated to a feature. Since all cards are played in all playouts, if we do not use features for MAST it will have uniform statistics for all the cards. So features are required for MAST. On the contrary of MAST, PPA compares the move in the same state and reinforces the played move, it will not have uniform statistics without features.

The feature we test for MASTF and PPAF is the depth of the move in the playout.

For MAST and PPA, we note MASTF and PPAF the versions with features.
For all the algorithms in the simulation, we take a number of internal simulations so that each algorithm has approximately the same running time (1 500 000 ms). We note ALGO(X) if we use ALGO with X internal simulations.

As expected, there is no huge differences between algorithms. However, we observe that MAST performs better than all the others.
PPA outperformed Flat and UCT, and UCT won against Flat.

4 Conclusion

To solve the game of ”6 Nimmt!”, we presented 4 different algorithms. They are all doing very well against a random player, approaching 100% of winrate.
For UCT, PPA and MAST, tuning the c, α and k constants is important and highly changes the performances.
Playing each algorithm against each other using the duplicate bridge method, MAST

Player 1	Player 2	Defense	Winner	Winrate winner
UCT (1500)	Flat (200)	MASTF (1000)	**UCT (1500)**	64%
MASTF (1000)	Flat (200)	MASTF (1000)	**MASTF (1000)**	64,3%
PPAF (1000)	Flat (200)	MASTF (1000)	**PPAF (1000)**	62,2%
UCT (1500)	MASTF (1000)	MASTF (1000)	**MASTF (1000)**	52,7%
PPAF (1000)	MASTF (1000)	MASTF (1000)	**MASTF (1000)**	53,7%
UCT (1500)	PPAF (1000)	MASTF (1000)	**PPAF (1000)**	50,2%
MASTF (1000)	MAST (1000)	MASTF (1000)	**MASTF (1000)**	57,1%
PPAF (1000)	PPA (1000)	MASTF (1000)	**PPAF (1000)**	55,7%

Fig. 7: Comparison of different algorithms playing against the same agent, for 300 games that have different results.

outperformed Flat, UCT and PPA.
PPA outperformed Flat and UCT, and Flat lost against all the other algorithms.

References

1. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on **4**(1), 1–43 (Mar 2012). https://doi.org/10.1109/TCIAIG.2012.2186810
2. Cazenave, T.: Playout policy adaptation with move features. Theoretical Computer Science **644**, 43–52 (2016)
3. Cazenave, T., Ventos, V.: The $\alpha\mu$ search algorithm for the game of bridge. In: Monte Carlo Search at IJCAI (2020)
4. Finnsson, H., Björnsson, Y.: Learning simulation control in general game-playing agents. In: AAAI. pp. 954–959 (2010)