Explanation of Nested Monte Carlo Search in games

Nina Braunmiller
Johannes Kepler Universität Linz
Linz, Austria
braunmiller.nina@tutanota.com

***Abstract*—The aim of this paper is it to explain the "Nested Monte Carlo Search" (Cazenave et al., 2016, p.687) (NMCS) based on the original paper "Nested Monte Carlo Search" written by Cazenave et al. (2016). Therefore, this work will firstly explain the basic idea of NMCS and its connection to Monte Carlo Tree Search (MCS). Then it will go into detail of improvements of the NMCS which are needed for the extension from one- to two-player games. Last but not least it will discuss the found results of the original paper.**

Keywords—Nested Monte Carlo Tree Search, Nested Monte Carlo Search, Monte Carlo Tree Search, two-player games, discounting, pruning

I. Introduction

NMCS seems to work rather good in single-player games. This was shown by the paper of Cazenave (2009). NMCS managed to make an world record in the single-player game "Morphin Solitaire disjoint version" (Cazenave, 2009, p. 459) (Cazenave, 2009). Furthermore, NMCS was rather good in the game "16x16 Sudoku" (Cazenave, 2009, p. 460) . Let's shortly explain the rules of Sudoku. We have a 16x16 grid which consists of smaller boxes of which each contains 4x4 entries which have to be filled with numbers ranging from 1 to 16. In the beginning some values are already placed in these boxes. So, each box shall contain each value exactly one time in the end. Furthermore, each value shall be unique in each row and column of the 16x16 grid (Cazenave, 2009; Easybrain, n.d.). It follows that the NMCS has to choose for each entry one value. The current entry will be always the one which has the smallest set of possible numbers because the occupied ones were already used in the related row and column and were therefore removed from the set of possible numbers. The more values were assigned to the empty entries the deeper is the level in the game tree. Therefore, current tree depth is used as evaluation value. The deeper the level the closer is the NMCS to a solution. Moreover, the best sequence of moves can be saved such that future execution will be much faster and the search will be improved (Cazenave, 2009).

However, it is not easy to transfer the advantages of NMCS in single-player games to two-player games. 1. From the above it becomes clear the the single-player NMCS prefers long paths within the game tree whereas the two-player NMCS wants to win as fast as possible. 2. Because depth is the value of the single-player NMCS we would observe a wide range of possible final values. In the two-player NMCS exists only the set {-1/loss, 0/tie, +1/win}. The question arises which losing/winning path is better than like-minded ones. 3. The best sequence of moves is saved to economize execution time in single-player NMCS but in the two-player version the opponent moves aren't predictable such that there is an lack of information and therefore no movement path can be built up. 4. Finally, the agent has unlimited time for decisions in one player mode which is not given for two-player games. With each additional explored level the execution costs increase exponentially (Cazenave et al., 2016).

It follows the question how to transfer the single-player NMCS into a two-player NMCS with acceptable execution time and keeping performance. In the following we will shortly talk about the MCS and extend it to the NMCS.

II. Nested monte carlo search

A. Monte Carlo Tree Search

To understand NMCS we first have to look at the MCS (Fig. 1). The procedure expires like that:

1. In the game tree select one node which represents the resulting state of a certain move/edge (Roy, 2019).
2. Expand the node by all its children and select one of these. This child will be called playout node "P" in this paper (Roy, 2019).
3. From this node P the algorithm randomly selects a sequence of nodes until it reaches a terminal node "T" with a value (-1/loss, 0/tie, +1/win) (Roy, 2019).
4. Backpropagate this value back starting from node P going upwards to the root of the game tree (Roy, 2019).

Note that the NMCS will only differ in the third step, the playout function.

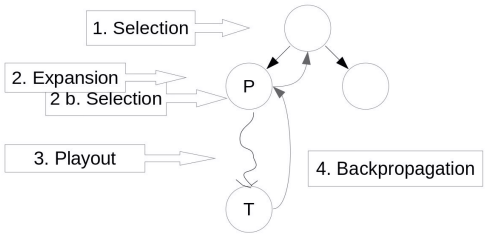

Fig. 1. Expire of a MCS in a game tree starting with the selected node.

B. Algorithmic understanding of Nested Monte Carlo Search

When talking about NMCS the crucial part is the nesting level "n". To go into detail: The playout function NMC(n) with

n=0 represents the MCS as described above. As soon as we increase n we get

$$NMC(n+1) = P_{\pi(V(NMC(n)))} \quad (1)$$

where P_π describes the playout function to get a sequence of states according to policy π which tries to find the best path starting from node P. V(NMC(N)) stands for an evaluation function that scores the value of a terminal node T that was reached by using NMC(n). It follows that the algorithm replaces the third step of the MCS procedure described above by using NMC(n) instead of simply choosing a random child. This behavior is repeated until NMC(0) was used (recursive structure; with each iteration n=n-1) (Cazenave et al., 2016).

To fully understand the NMCS algorithmically have a look at the upcoming series of figures. With its help the algorithm of NMCS shall be explained step by step.

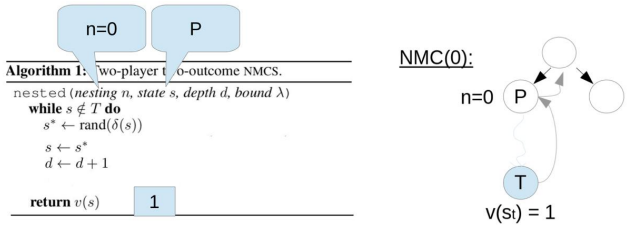

Fig. 2. MCS=NMC(0) as a game tree with its policy as pseudo code (Cazenave et al., 2016). The blue markings show which parts are important for this figure.

In Fig. 2 you can see the MCS and its policy as pseudo code. It shows us that ongoing from node P the algorithm randomly selects a sequence of moves which end up in a terminal node T with an value v=+1 in this example (Cazenave et al., 2016).

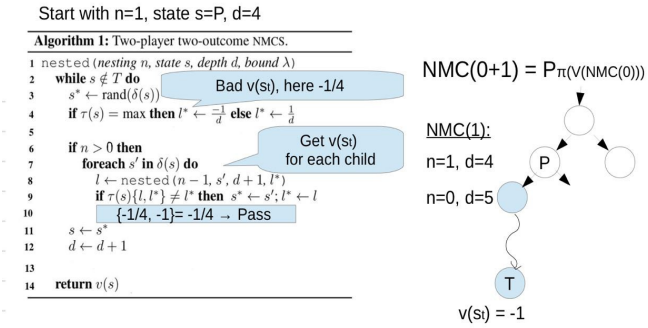

Fig.3. NMC(1): pseudo code refers to the policy function (Cazenave et al., 2016); the game tree clarifies an example situation. The blue markings show which parts are important for this figure.

Fig. 3 shows an extension of the pseudo code of Fig. 2. Line 4 initializes l* which would be a bad value which we want to top. Line 4 says: When we have an MAX player (the player that tries get the highest final value out of a game) then

$$l^* = \frac{-1}{d_s} \quad (2)$$

with "d_s" as the depth of the current state. However, MAX wants to reach a node T with v>0 and therefore v>l*. When the current nesting level n>0, the algorithm looks at all the children of the current node by using recursion with n=n-1 for iterations. When n=0, then randomly select the children until T is reached. In Fig. 3 it's v=-1. The MAX player compares the "-1" with the l* (here -1/4). It would choose l* because it is closer to +1/win. Therefore, nothing happens (Cazenave et al., 2016).

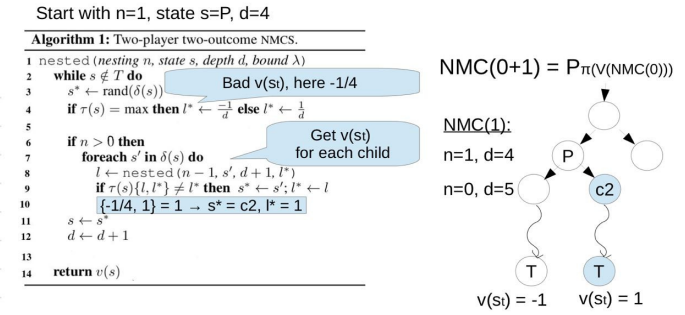

Fig. 4. NMC(1): continuation of Fig. 3 which concentrates on the second child "c2" of node P. With this adaption the NMC(1) becomes complete in its policy (Cazenave et al., 2016). The blue markings show which parts are important for this figure.

Looking at the next child "c2" of P we get node T with v=+1 (Fig. 4). This value is closer to a +1/win than l*=-1/4. Therefore, we would select the path along c2 and update l*=+1. When there are other children of node P they would have to top the updated l* (Cazenave et al., 2016).

However: How can +1/win be topped? Are there wins which a more preferable than other wins? That leads us to the next chapter of this paper.

III. Adaptions of Nested Monte Carlo Search to Two-Player Games

The aim was it that the NMCS for two-player games can be executed in an appropriate time and delivers good performance.

A. *"Discounting"* (Cazenave et al., 2016, p. 688)

The player wants to win as fast as possible in a two-player game because the opponent would have less opportunities of escaping in that way. Therefore, the player also wants to lose as slow as possible such that he/she/it can find a way out of losing (Cazenave et al., 2016).

Discounting: That's why, a value v of the node T can be expressed as

$$v_D(P_\pi, s) = \frac{v(s_t)}{t+1} \quad (3)$$

where s_t is the current node and (t+1) is the number of moves needed to reach a node T beginning from s_t (Cazenave et al., 2016). It follows, that value v_D of node T which can be only reached with a lot of moves converges to +0 when v=+1 and -0 when v=-1. When v_D is reached in few steps, it comes closer to +1 in a winning case (v=+1) and -1 when losing (v=-1).

Because +0<+1, $v_D \approx +0$ (almost a tie) wouldn't be desirable, hence choose winning nodes T which are closer to s_t.

Furthermore, we can observe that -0>-1. It follows that -0 is closer to +1/win than -1. Therefore, prefer the long path ending with $v_D \approx -0$ when no winning path can be found.

It follows that the algorithm pepped up with discounting is better in move selection. However, discounting hasn't impact on the complexity (Cazenave et al., 2016). That's why pruning, which is explained in the next section, is needed.

B. *Pruning*

Pruning means that the algorithm tactically doesn't visit all nodes in the game tree (Cazenave et al., 2016). Also here only the playout phase matters. In Fig. 4 became clear that all children of node P are visited. For each child in NMC(1) we randomly selected a sequence of nodes until T was reached. For explanation reasons the following makes use of NMC(1).

1) "Pruning on Depth" (Cazenave et al., 2016, p. 689) *(POD)*

With POD we also look at all the children of node P. We also follow the sequence of moves until node T is reached. However, we can also interrupt this search. For this we also consider discounting. When we find the first time a node T with value +1/win we save its v_D. Whenever an other path of an other child of node P leads with less steps to a +1/win we get a higher v_D. Then the saved v_D is overwritten. Whenever a path of an other child of node P becomes longer than the depth of the saved winning node T with the best v_D, we stop searching and go to the next child (Cazenave et al., 2016). The reason for this is that this path couldn't beat the saved path with the best v_D because more steps would be needed. Also have a look at Fig. 5(a).

2) "Cut on Win" (Cazenave et al., 2016, p. 689) *(COW):*

Whenever the algorithm reaches a node T which with v=+1, it stops searching (Cazenave et al., 2016). That means all children of node P which weren't checked so far are simply ignored. From this it becomes clear that discounting isn't considered because the algorithm doesn't distinguish in the quality of different winning nodes T. Therefore, better move selection isn't taking into account. However, complexity is decreased drastically. Also have a look at Fig. 5(b).

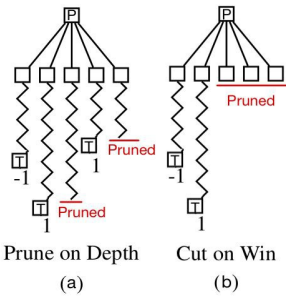

Fig. 5. NMC(1): game tree starting from node P. For normal the algorithm would look at all P's children. This isn't the case in (b). In (a) the algorithm visits all children but stops the search in a branch when its path becomes too long (Cazenave et al., 2016).

IV. Remarks on the Results

The scientists (Cazenave et al., 2016) tested the NMCS on five different two-player games and in four of these also their misère version which is the same game with reverse winning conditions. For all games they run 500 matches between MCS and NMCS and took the precentage of wins for each side. When we have a look at the results of the longest allowed decision time (320 ms) we can observe the following. In eight of the nine different game versions NMCS with n=1 and COW achieved the highest winning scores against MCS (Cazenave et al., 2016). This is astonishing because through COW the move quality gets ignored. A reason for that observation could be that a move decision time of 320 ms could be too short for the NMCS with n=1 and POD. That would also explain why NMCS with n=2 and COW performs worse than with n=1 (Cazenave et al., 2016).

Let's only consider this NMCS with n=1 and COW. In five cases the NMCS won clearly more often against MCS than vice versa. In three games both NMCS and MCS won around half of the matches. The reason could be that these games are quite easy and one game colour leads to a win in perfect play. This colour was played 250 times per algorithm. In only one game MCS outperforms NMCS (Cazenave et al., 2016).

V. Conclusions

The NMCS is an extension of MCS which only differs in the playout policy of the game tree. The transfer from NMCS for single-player games to NMCS for two-player games worked out.

Discouting in combination with POD is a handy tool for better move selection and avoiding too high search complexity. However, in practice isn't so much time of computation. Therefore, NMCS with n=1 and COW succeeds. It would be interesting to see results for a decision time > 320 ms.

In total, NMCS can outperform MCS under certain conditions (execution time, game, nesting level n, pruning method) (Cazenave et al., 2016).

References

[1] Cazenave, T. (2009, July). Nested Monte-Carlo Search. *IJ-CAI*, 456-461. https://www.ijcai.org/Proceedings/09/Papers/083.pdf

[2] Cazenave, T., Saffidine, A., Schofield, M., & Thielscher, M. (2016, February 21). Nested Monte Carlo Search for Two-Player Games. *AAAI-16*, 687-693. https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12134/11652

[3] Easybrain. (n.d.). *How to Play Sudoku?* sudoku.com. Retrieved July 06, 2021, from https://sudoku.com/how-to-play/sudoku-rules-for-complete-beginners/

[4] Roy, R. (2019, January 14). *ML | Monte Carlo Tree Search (MCTS)*. geeksforgeeks.org. Retrieved July 06, 2021, from https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/