Nested Monte Carlo Search for Two-player Games

Tristan Cazenave
LAMSADE — Université Paris-Dauphine
cazenave@lamsade.dauphine.fr

Abdallah Saffidine
Michael Schofield
Michael Thielscher
School of Computer Science and Engineering
The University of New South Wales
{abdallahs, mschofield, mit}@cse.unsw.edu.au

Abstract

The use of the Monte Carlo playouts as an evaluation function has proved to be a viable, general technique for searching intractable game spaces. This facilitate the use of statistical techniques like Monte Carlo Tree Search (MCTS), but is also known to require significant processing overhead. We seek to improve the quality of information extracted from the Monte Carlo playout in three ways. Firstly, by *nesting* the evaluation function inside another evaluation function; secondly, by measuring and utilising the depth of the playout; and thirdly, by incorporating pruning strategies that eliminate unnecessary searches and avoid traps. Our experimental data, obtained on a variety of two-player games from past General Game Playing (GGP) competitions and others, demonstrate the usefulness of these techniques in a Nested Player when pitted against a standard, optimised UCT player.

Introduction

Monte Carlo techniques have proved a viable approach for searching intractable game spaces (Browne et al. 2012). These techniques are domain independent, that is, the programmer does not need to construct a different evaluation function for each game. As such, they have been successfully applied to a variety of games, especially in General Game Playing (GGP), where the goal is to build AI systems that are capable of taking the rules of any game described in a formal language and to play that game efficiently and effectively (Genesereth, Love, and Pell 2005). This area of research is growing with much emphasis on improving the ability of AI systems to play games with intractable search spaces.

A policy is a function or algorithm mapping states to actions for a specified player. The simplest domain independent policy is to always choose a random move, but this does not lead to quality play. This raises a fundamental question: how do we take a simple random technique and improve the quality of the information it produces while mitigating the increase in complexity, for example in the context of GGP?

For the special case of single-agent games, the Nested Monte Carlo Search (NMCS) algorithm was proposed as an alternative to single-player Monte Carlo Tree Search (MCTS) (Cazenave 2009). It lends itself to many optimisations and improvements and has been successful in many such problems (Cazenave 2010; Akiyama, Komiya, and Kotani 2012). In particular, NMCS lead to a new record solution to the Morpion Solitaire mathematical puzzle.

While the NMCS algorithm has been used successfully in a variety of single-agent domains, it has not been adapted to a more general setting. The purpose of our paper is to generalise the NMCS algorithm to two-player turn-taking games with win/lose outcomes.

Contributions

A naive extension of NMCS from single- to two-player games faces three hurdles: (a) no game theoretic guarantees, (b) lack of informativity of the Monte Carlo playouts, and (c) high complexity of nested searches. Our contributions in this paper address each of these issues as follows: Based on a formal framework for NMCS in two-player win/lose games,

1. we show how the quality of information propagated during the search can be increased via a *discounting heuristic*, leading to a better move selection for the overall algorithm;
2. we present a technique for improving the cost-effectiveness of the algorithm without changing the resulting policy by using *safe pruning* criteria;
3. we show how long-term convergence to an optimal strategy can be guaranteed by wrapping NMCS inside a Upper Confidence bounds for Trees (UCT)-like algorithm;

Moreover, we present the results of evaluating experimentally the performance of the algorithm on 9 different two-player win/lose games. The player resulting from combining these ideas is superior to a standard MCTS player in 5 of the tested domains, and weaker in a single one. This NMCS player also performs favourably compared to the recently developed MCTS-MR algorithm (Baier and Winands 2013) in 4 domains and is outperformed in 2. We identify *misère* variants as being particularly favourable to two-player NMCS.

The remainder of the paper is organised as follows. We introduce two-player, two-outcome games and a generalisation of single-player NMCS to these games, then present the two heuristic improvements of discounting and pruning, respectively, including game-theoretic guarantees, and finally state and discuss our experimental results.

Nested Monte Carlo Search

We now describe two-player two-outcome games. These games have two possible type of terminal states, labelled -1 and 1. The minimizing player is trying to reach a -1 state while the maximizing player tries to end in a 1 terminal state.

Formally, a *game* is a tuple $\langle S, T, v, \delta, \tau \rangle$ where S is a set of *states* in the game, $T \subseteq S$ is the set of *terminal states*, and $D = S \setminus T$ is the set of *decision states*. $v : T \to \{-1, 1\}$ is the *payoff* function on termination. $\delta : D \to 2^S$ is the *successor* function, i.e., $\delta(s)$ is the set of states reachable from s in one step. $\tau : D \to \{\min, \max\}$ is the *turn* function, indicating which role choose the next action.

A move selection *policy* is a mapping from decision states to distribution probabilities across states, $\pi : D \to \phi(S)$.[1] Policies can only select successor states, so for any state $d \in D$ and any policy π, the support of $\pi(d)$ is contained in $\delta(d)$. The *uniform policy* returns any successor state with the same probability.

A *playout* is a sequence of successive states ending in a terminal state, i.e., for $s_0 s_1 \dots s_t$ we have $s_{i+1} \in \delta(s_i)$ and $s_t \in T$. Let π be a policy, the *playout function of* π, P_π, maps input states to sequences of states drawn according to π. That is, for any state $s \in S$, $P_\pi(s) = s_0 \dots s_t$ where $s_0 = s$ and $s_{i+1} \sim \pi(s_i)$. Note that we could also define playout functions using a different policy for each role, but it is not required for the rest of the paper. The *Monte Carlo playout* is a playout function of the *uniform policy*.

A (stochastic) *evaluation function* is a mapping from states to (distributions over) real values. Let P_π be the playout function of some policy π. The *evaluation function of* P_π, written $V(P_\pi)$, maps states to the payoff of the terminal state reached with P_π from these states. For example, for a state s, if $P_\pi(s)$ returns $s_0 \dots s_t$, then $V(P_\pi, s)$ would return $v(s_t)$. If π is a stochastic policy, then different playouts can result from the same starting state, different rewards can be obtained, and $V(P_\pi)$ is thus stochastic as well.

Conversely, given an evaluation function f, it is possible to build a corresponding policy $\Pi(f)$ by choosing the successor maximizing (resp. minimizing) the evaluation function on max (resp. min) decision states. This duality between policies and evaluation functions is the main intuition behind the NMCS algorithm.

Definition 1. For any nesting level n, $\text{NMC}(n)$ is a playout function that maps states to playouts. We define it by induction as follows: $\text{NMC}(0)$ is the Monte Carlo playout function, and $\text{NMC}(n+1) = P_{\Pi(V(\text{NMC}(n)))}$.

For example, to obtain an $\text{NMC}(2)$ playout from an input state s, the algorithm first evaluates each of the $|\delta(s)|$ successor states by playing out a full game using an $\text{NMC}(1)$ playout for each role. Each $\text{NMC}(1)$ playout would use Monte Carlo playouts for each move choice for the length of the game. After the $\text{NMC}(1)$ results have been gathered for the successors of s, the algorithm chooses a successor state by maximizing or minimizing over these results depending on the turn player. This procedure is iterated until a terminal state is reached by the main procedure.

[1] The notation $\phi(S)$ denotes the set of distributions over S.

It is easy to see that the computational cost of this algorithm grows exponentially with the nesting level. Two ideas allow to improve the cost-effectiveness of two-player NMCS. Using the playout depth for *discounting* increases the evaluation function quality without increasing cost; Safe search *pruning* reduces the cost of the evaluation function without reducing its quality.

Heuristic Improvement I: Discounting

The naive porting of NMCS to two-outcome games described in the previous section results in only two classes of moves at each search nodes: moves that have lead to a won sub-playout and moves that have lead to a lost sub-playout. One of the main differences between two-outcome two-player games and the single-agent domains in which NMCS is successful is that the latter offer a very wide range of possible game outcomes. These different game outcomes help distinguish moves further.

The discounting heuristic turns a win/loss game into a game with a wide range of outcomes by having the max player preferring short wins to long wins, and long losses to short losses. The intuition here is to win as quickly as possible so as to minimize our opponent's chance of finding an escape, and to lose as slowly as possible so as to maximize our chance of finding an escape.

This idea can be implemented by replacing the V operator in the definition of $\text{NMC}()$ by a discounting version V_D. The new V_D operator is such that if $P_\pi(s)$ returns $s_0 \dots s_t$, $V_D(P_\pi, s)$ would return $\frac{v(s_t)}{t+1}$. This way, the ordering between game outcomes corresponds exactly to the ordering between the scores considered as real numbers. We call the resulting playout function $\text{NMC}_D()$.

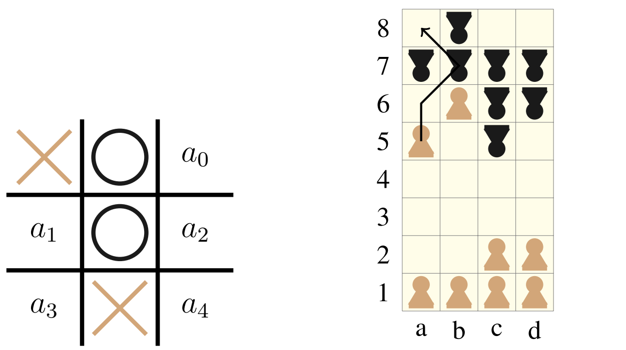
While our application of discounting playouts to NMCS is new, the idea has already appeared for MCTS in two flavours. Just like us, Finnsson and Björnsson (2008) discount on the length of the playout whereas Steinhauer (2010) discounts on how long ago a playout was performed. Nevertheless, Two important differences exist between Finnsson and Björnsson (2008)'s approach and ours. First, theirs addresses scores ranging from 0 to 100 while ours use $\{-1, 1\}$. As a result, long and short losses are not treated differently in their work. Second, discounted rewards only affect the selection in the NMCS part of our algorithm and full loss/win results are propagated in the UCT tree.

Game-theoretic guarantees

A close examination of the tree of nodes reached during an $\text{NMC}(n)$ or $\text{NMC}_D(n)$ call shows that it contains small full minimax trees of depth n rooted at each node of the returned playout. In particular, a minimax search has been performed in the first node of the playout and the following propositions can be derived easily.

Proposition 1. *If there is a forced win or loss in n moves from a state s, the value $V(\text{NMC}(n), s)$ returned by the NMCS algorithm with nested level n is correct.*

Proposition 2. *If there is a forced win or loss in $n+1$ moves from a state s, the move $\Pi(V_D(\text{NMC}_D(n)), s)$ recommended*



(a) The X player is to play. Any move except a_3 leads to a draw with perfect play.

(b) White's winning move is `a5-a6`. `b6-a7` and `b6-c7` initially seem good but are blunders.

Figure 1: Partially played games of TicTacToe and Breakthrough with a single winning move for the turn player.

Table 1: Effect of discounting on the distribution of nested level 2 policies applied to Figure 1a, across 1000 games.

Move	$\Pi(V(\text{NMC}(2)))$		$\Pi(V_D(\text{NMC}_D(2)))$	
	Value	Frequency	Value	Frequency
a_0	$\{0\}$	0	$\{0\}$	0
a_1	$\{0, 1\}$	176	$\{0\}$	0
a_2	$\{0, 1\}$	123	$\{0, \frac{1}{4}\}$	0
a_3	$\{1\}$	575	$\{\frac{1}{2}\}$	1000
a_4	$\{0, 1\}$	126	$\{0, \frac{1}{4}\}$	0

by the NMCS algorithm with nested level n and discounting is optimal.

Proposition 1 holds whether discounting is used or not but Proposition 2 does not hold in general for NMCS without discounting. Figure 1a provides a counter-example. Player X has a theoretical win in 3 in this Tictactoe position and a_3 is the only correct move. Table 1 summarizes the data obtained from calling $\Pi(V(\text{NMC}(2)))$ and $\Pi(V_D(\text{NMC}_D(2)))$ a 1000 times on this position. *Value* indicates the set of encountered values across all the calls, and *Frequency* indicates how often the corresponding move has been selected by the policy.

Without discounting, moves a_1, a_2, and a_4 are selected occasionally in a tie break because they sometimes lead to a won playout. With discounting, however, the shorter (and guaranteed) win of a_3 is always preferred to those occuring after a_2 and a_4. Note also that when discounting is used, the embedded $\text{NMC}_D(2)$ selecting moves for the O player is forcing a draw after move a_1.

Heuristic Improvement II: Pruning

From the classical alpha-beta search (Knuth and Moore 1975) to the more recent Score-Bounded MCTS adaptation (Cazenave and Saffidine 2010), two-player games usually lend themselves quite well to pruning strategies. Adapting NMCS to two-player games also gives pruning opportunities that were absent in the single-player case.

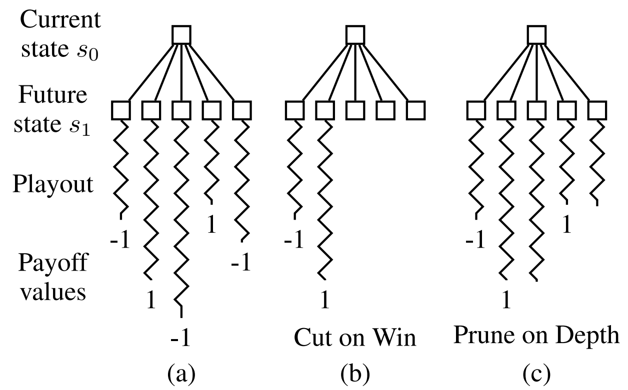


Figure 2: Effect of the pruning strategies on an NMCS run. We assume a max root state and a left-to-right evaluation order. (a) In the standard case, the second and the fourth successor states are equally preferred. With discounting, the fourth successor state is the most-preferred one. (b) This fourth state may fail to be selected when Cut on Win is enabled. (c) With Pruning on Depth and discounting, however, this fourth state would be found and preferred too.

Cut on Win The first pruning strategy, Cut on Win (COW), hangs on two features of two-player win/loss games. First, we know the precise value of the best outcome of the game. Second, these values are reached regularly in playouts.

The COW strategy consists of evaluating the moves randomly and selecting the first move that returns the maximum value from its evaluation. This can be achieved through replacing Π by a pruning version Π_{COW}. Specifically, $\Pi_{\text{COW}}(f, s)$ would call f on one successor state of s at a time, until one with a positive value is found in case $\tau(s) = \max$, or with a negative value in case $\tau(s) = \min$, and then discard all the remaining successors.

Proposition 3. *Assume that whenever a state is evaluated using a nested playout, successor states are presented in a random order. Then $\text{NMC}(n)$ with COW generates the same playout distribution as $\text{NMC}(n)$ with no pruning.*

Proposition 3 relies on no discounting being used. In general, $\text{NMC}_D(n)$ with COW is not guaranteed to have the same distribution as $\text{NMC}_D(n)$, as illustrated in Figure 2(a)–(b).

Pruning on Depth The second pruning strategy, Prune on Depth (POD), takes into account the richer outcome structure offered by the discounting heuristic. In POD, the search does not return at the first occurrence of a win, but whenever a playout is bound to be longer than an already explored won sibling. Put simply, in max (resp. min) states we prune states deeper than the shallowest win (resp. loss), as depicted in Figure 2(c). A safety result similar to that of COW can be derived.

Proposition 4. *Assume that whenever a state is evaluated using a nested playout, move choices are randomly presented in the same order. Then $\text{NMC}_D(n)$ with no pruning returns the same move as $\text{NMC}_D(n)$ with POD.*

Table 2: Performance of NMC(3) and $\text{NMC}_D(3)$ starting from Figure 1b, averaged over 900 runs; showing how discounting and pruning affect the number of states visited and the correct move frequency.

Discounting	Pruning	States Visited(k)	Freq(%)
No	None	$4,459 \pm 27$	11.9 ± 2.2
No	COW(≤ 1)	$1,084 \pm 8$	12.3 ± 2.6
No	COW(≤ 2)	214 ± 2	10.9 ± 2.0
No	COW(≤ 3)	25 ± 1	9.8 ± 2.0
Yes	None	$2,775 \pm 26$	64.1 ± 3.4
Yes	POD(≤ 1)	$1,924 \pm 20$	64.7 ± 3.5
Yes	POD(≤ 2)	$1,463 \pm 16$	58.6 ± 3.5
Yes	POD(≤ 3)	627 ± 19	62.4 ± 3.3

The Trade-off of Unsafe Pruning

Unlike the classical alpha-beta algorithm, the COW technique described previously is *unsafe* when using discounting: it may lead to a better move being overlooked. Unsafe pruning methods are common in the game search community, for instance *null move* and *forward pruning* (Smith and Nau 1994), and the attractiveness of a new method depends on the speed versus accuracy trade-off.

As an illustrative example, we look at the game of Breakthrough being played in Figure 1b.[2] This position was used as a test case exhibiting a trap that is difficult to avoid for a plain UCT player (Gudmundsson and Björnsson 2013). The correct move for White is `a5-a6`, however `b6-a7` and `b6-c7` initially look strong.

We generate multiple NMC(3) and $\text{NMC}_D(3)$ playouts with different parameter settings, starting from this position. For each parameter setting, we record the number of states visited and the likelihood of selecting the correct initial move and present the results in Table 2. A setting of the type COW($\leq i$) is to be understood as Cut on Win pruning heuristic was activated at nesting levels i and below but not at any higher level. Each entry in the table is an average of 900 runs, and the 95% confidence interval on standard error of the mean is reported for each entry.

Observe from Table 2 that (a) Discounting improves the likelihood of finding the best move from 0.11 to 0.63; (b) all pruning strategies significantly reduce the search effort; (c) the performance of a non-discounted (resp. discounted) search is not significantly affected by COW (resp. POD), as predicted by Proposition 3 (resp. 4); and (d) POD(≤ 3) is 25 times as expensive as COW(≤ 3), but much more accurate.

The quality of the choices being made by the level 3 policy is built on the quality of the choices being made by the embedded level 2 player for both Black and White roles. Remember that the move `b6-a7` is a trap for White as it eventually fails, but the Black level 2 player must spring the trap for the White level 3 playout to "get it right".

[2] Knowing the rules of Breakthrough is not essential to follow our analysis. However, the reader can find a description of the rules in related work (Saffidine, Jouandeau, and Cazenave 2011; Gudmundsson and Björnsson 2013).

Black must respond with `b8-a7`, otherwise White has a certain win. Our experiments indicate that an NMC(2) player selects `b8-a7` 15% of the time, whether COW is enabled or not, whereas an $\text{NMC}_D(2)$ player selects this move 100% of the time, whether POD is enabled or not.

Algorithm

As a summary, Algorithm 1 provides pseudo-code for our generalisation of the NMCS algorithm to two-player games. Lines 10, 5, and 13 respectively allow to enable the cut on win, pruning on depth, and discounting heuristics.

Algorithm 1: Two-player two-outcome NMCS.

1 `nested`(*nesting* n, *state* s, *depth* d, *bound* λ)
2   **while** $s \notin T$ **do**
3     $s^* \leftarrow \text{rand}(\delta(s))$
4   **if** $\tau(s) = \max$ **then** $l^* \leftarrow \frac{-1}{d}$ **else** $l^* \leftarrow \frac{1}{d}$
5   **if** *d-pruning* **and** $\tau(s)\{-l^*, \lambda\} = \lambda$ **then return** λ
6   **if** $n > 0$ **then**
7   **foreach** s' **in** $\delta(s)$ **do**
8         $l \leftarrow$ `nested`($n-1$, s', $d+1$, l^*)
9   **if** $\tau(s)\{l, l^*\} \neq l^*$ **then** $s^* \leftarrow s'$; $l^* \leftarrow l$
10   **if** *cut on win* **and** $\tau(s)\{l, 0\} \neq 0$ **then break**
11     $s \leftarrow s^*$
12     $d \leftarrow d + 1$
13   **if** *discounting* **then return** $\frac{v(s)}{d}$
14   **else return** $v(s)$

Experimental Results

Domains

We use 9 two-player games drawn from games commonly played in GGP competitions, each played on a 5×5 board.

Breakthrough and Knightthrough are racing games where each player is trying to get one their piece across the board, these two games are popular as benchmarks in the GGP community.

Domineering and NoGo are mathematical games in which players gradually fill a board until one of them has no legal moves remaining and is declared loser, these two games are popular in the Combinatorial Game Theory community.

For each of these domain, we construct a *misère* version, which has exactly the same rules but with reverse winning condition. For instance, a player wins misère Breakthrough if they force their opponent to cross the board.

To this list, we add AtariGo, a capturing game in which each player tries to surround the opponent's pieces. AtariGo is a popular pedagogical tool when teaching the game of Go.

Performance of the playout engine

It is well known in the games community that increasing the strength of a playout policy may not always result in a strength increase for the wrapping search (Silver and Tesauro 2009). Still, it often is the case in practice, and determining whether our *discounting* heuristic improves

Table 3: Winrates (%) of NMCS with discounting vs. NMCS without it for nesting levels 0 to 2 and game engine speed.

Game	Nesting Level			States visited
	0	1	2	per second (k)
Breakthrough	79.6	99.6	99.4	411
misère	42.4	80.8	90.0	409
Knightthrough	78.6	100.0	100.0	264
misère	46.0	83.2	85.8	328
Domineering	71.2	77.0	83.8	550
misère	43.4	63.2	68.4	592
NoGo	62.8	76.4	83.4	357
misère	53.2	65.6	67.2	648
AtariGo	69.6	97.2	100.0	280

the strength of nested playout policies may prove informative. For each of the nine domains of interest, and for each level of nesting n from 0 to 2, we run a match between $\Pi(V_D(\text{NMC}_D(n)))$ and $\Pi(V(\text{NMC}(n)))$. 500 games are played per match, 250 with each color, and we provide the winrates of the discounting nested player in Table 3.

The performance of the wrapping search may also be affected by using a slower playout policy. Fortunately, discounting does not slow NMCS down in terms of states visited per second, and preliminary experiments revealed that discounting even decreases the typical length of playouts, thereby increasing the number of playouts performed per second. As a reference for subsequent fixed-time experiments, we display the game engine speed in thousands of visited states per second in the last column of Table 3. The experiments are run on a 3.0 GHz PC under Linux.

Parameters and Performance against UCT

We want to determine whether using nested rather than plain Monte Carlo playouts could improve the performance of a UCT player in two-player games in a more systematic way. We also want to measure the effect of the heuristics proposed in the previous section.

We therefore played two versions of MCTS against each other in a variety of domains. One runs the UCT algorithm using nested playouts (labelled NMCS) and the other is a standard MCTS, *i.e.*, an optimised UCT with random playouts. Both players are allocated the same thinking time, ranging from 10ms per move to 320ms per move. We try several parameterisation of NMCS: nesting depth 1 or 2, COW, and the combination of discounting and POD. For each game, each parameter setting, and each time constraint, we run a 500 games match where NMCS plays as first player 250 times and we record how frequently NMCS wins in Table 4.

The first element that we can notice in Table 4 is that both discounting and COW improve significantly the performance of NMCS over using level 1 playouts with no heuristics. We can also observe that for this range of computational resources, using a level 2 nested search for the MCTS playouts does not seem as effective as using a level 1 nested search.

In two domains, Knightthrough and Domineering, the winrate converges to 50% as the time budget increases.

Table 4: Win percentages of NMCS against a standard MCTS player for various settings and thinking times.

Game		n	COW	POD	10ms	20ms	40ms	80ms	160ms	320ms
Breakthrough		1			3.2	6.0	12.0	11.6	7.8	6.4
		1		✔	27.6	22.6	16.8	21.6	15.4	20.4
		1	✔		22.6	25.2	30.4	34.6	35.2	39.6
		2	✔		4.6	2.0	2.4	1.4	2.4	3.8
	misère	1			85.4	83.4	70.2	60.8	57.0	56.4
		1		✔	91.4	95.6	97.0	97.8	98.8	98.8
		1	✔		95.2	95.2	98.0	99.0	99.8	**99.8**
		2	✔		1.0	27.6	43.6	87.0	93.2	95.6
Knightthrough		1			42.2	57.2	9.8	49.4	50.2	50.0
		1		✔	68.6	50.2	42.4	42.4	46.4	44.6
		1	✔		27.2	25.4	28.0	43.4	49.2	49.6
		2	✔		20.0	16.4	5.8	1.8	29.2	38.2
	misère	1			43.0	31.6	20.0	15.4	11.2	12.6
		1		✔	54.6	72.2	80.6	88.4	94.2	98.4
		1	✔		77.8	82.2	88.8	94.4	98.2	**98.6**
		2	✔		20.8	18.6	32.2	42.2	54.0	67.0
Domineering		1			13.4	8.6	8.6	6.0	14.2	28.0
		1		✔	40.8	34.4	37.4	48.4	50.0	50.0
		1	✔		44.4	38.6	40.6	49.4	50.0	50.0
		2	✔		11.2	14.4	20.2	25.2	32.2	45.4
	misère	1			33.4	25.2	20.0	18.8	13.2	12.2
		1		✔	45.4	47.2	56.8	60.2	62.8	54.2
		1	✔		69.4	66.6	71.6	70.4	68.4	**58.6**
		2	✔		37.0	45.2	45.6	51.0	57.8	53.6
NoGo		1			5.8	3.0	2.6	3.0	0.6	0.8
		1		✔	7.2	16.0	31.8	35.2	35.4	40.6
		1	✔		37.6	39.2	38.4	40.8	47.8	48.0
		2	✔		0.4	2.8	5.4	15.0	20.6	17.0
	misère	1			14.6	6.6	5.2	3.0	2.4	1.8
		1		✔	17.2	25.0	38.8	51.2	48.2	48.8
		1	✔		55.4	56.6	57.0	57.6	54.6	**60.8**
		2	✔		5.2	10.6	19.4	35.6	37.2	47.8
Atari-Go		1			0.6	2.2	4.6	5.4	6.8	7.6
		1		✔	0.2	19.2	42.0	42.0	55.4	67.2
		1	✔		42.0	59.0	60.2	71.0	71.2	**77.2**
		2	✔		0.2	0.0	0.6	7.4	8.6	4.8

Manual examination indicates that the same side wins almost every game, no matter which algorithm plays White. We interpret this to mean that Domineering and Knightthrough appear to be an easy task for the algorithms at hand. On the other hand, the large proportion of games lost by the reference MCTS player independent of the side played demonstrate that some games are far from being solved by this algorithm, for instance misère Breakthrough, misère Knightthrough, or AtariGo are dominated by NMCS. This shows that although all 9 games were played on boards of the same 5×5 size, the underlying decision problems were of varying difficulty.

The performance improvement on the misère version of the games seems to be much larger than on the original versions. A tentative explanation for this phenomenon which would be consistent with a similar intuition in single-agent domains is that NMCS is particularly good at games where the very last moves are crucial to the final score. Since the last moves made in a level n playout are based on a search of an important fraction of the subtree, comparatively fewer mistakes are made at this stage of the game than a plain Monte Carlo playout. Therefore, the estimates of a position's value are particularly more accurate for the nested playouts than for Monte Carlo playouts. This is consistent with the fact that NMCS is performing a depth-limited Minimax search at the terminus of the playout.

Comparison to MCTS-MR

The idea of performing small minimax searches in the playout phase of MCTS has been formally studied recently with the development of the MCTS with Minimax Rollouts (MCTS-MR) algorithm (Baier and Winands 2013). Previous work has shown that searches of depth 1 improve the performance of MCTS engines in Havannah (Lorentz 2010; Ewalds 2012), and searches of depth 2 improve the performance in Connect 4 (Baier and Winands 2013) and Lines of Action (Winands and Björnsson 2011).

NMCS explores small minimax trees at every node of a playout, but also creates sub-playouts. To determine whether the performance gains observed in Table 4 are entirely due to this minimax aspect, we compare the performance of NMCS with that of MCTS-MR. According to Table 4, the best parameter setting for NMCS is a nesting depth of $n = 1$ with COW pruning but no discounting. For MCTS-MR, we use searches of depth 1 and 2. In Table 5, the numbers represent the winrate percentage againt a standard MCTS opponent over 500 games.

Except in NoGo and misère NoGo, the MCTS-MR algorithm does not improve performance over standard MCTS. Therefore, our tentative explanation that the NMCS performance boost could be attributed to its avoiding late-game blunders via Minimax is only supported in misère NoGo. For the other misère games, it appears that improving the simulation quality at every step of the playouts is responsible for the particularly good performance of NMCS.

Table 5: Win percentages of NMCS and MCTS-MR against standard MCTS with 320ms per move. NMCS uses COW and depth 1 while MCTS-MR uses depth 1 and 2.

Game	NMCS	MCTS-MR	
		MR 1	MR 2
Breakthrough	39.6	47.4	**50.2**
misère	**99.8**	49.4	48.6
Knightthrough	49.6	50.0	50.0
misère	**98.6**	49.8	45.0
Domineering	50.0	50.0	49.8
misère	**58.6**	46.0	44.2
NoGo	48.0	**59.4**	50.2
misère	60.8	54.2	**67.4**
AtariGo	**77.2**	44.0	47.0

Discussion

The NMCS algorithm was proposed as an alternative to single-player MCTS for single-agent domains (Cazenave 2009). It lends itself to many optimisations and improvements and has been successful in many such problems (Cazenave 2010; Akiyama, Komiya, and Kotani 2012). In particular, NMCS lead to a new record solution to the Morpion Solitaire mathematical puzzle.

In this paper, we have examined the adaptation of the NMCS algorithm from single-agent problems to two-outcome two-player games. We have proposed two types of heuristic improvements to the algorithm and have shown that these suggestions indeed lead to better performance than that of the naive adaptation. In particular, discounting the reward based on the playout length increases the accuracy of the nested searches, and the various pruning strategies allow the discarding of very large parts of the search trees.

Together these ideas contribute to creating a new type of domain agnostic search-based artificial player which appears to be much better than a classic UCT player on some games. In particular, in the games misère Breakthrough and misère Knightthrough the new approach wins close to 99% of the games against the best known domain independent algorithm for these games.

In terms of related work, the intuition behind NMCS inspired the Nested Rollout Policy Adaptation algorithm which enabled further record establishing performances in similar domains (Rosin 2011). The idea of nesting searches of a certain type has also been used with MCTS to build opening books (Chaslot et al. 2009), with Proof Number Search to distribute it over a cluster (Saffidine, Jouandeau, and Cazenave 2011), and with Perfect Information Monte Carlo (PIMC) search as a way to alleviate the strategy fusion and non-local dependencies problems exhibited by PIMC in imperfect information games (Furtak and Buro 2013).

Méhat and Cazenave (2010) compare NMCS and the UCT algorithm for single player games with mixed results. They explore variants of UCT and NMCS and conclude that neither one is a clear winner. Pepels et al. (2014) have shown in the context of MCTS that more information than a binary outcome could be extracted from a random playout, even when very little domain knowledge is available. In particular, the outcome of a short playout might be more informative than that of a longer one because fewer random actions have taken place.

Some important improvements to the original single-player NMCS algorithm such as *memorization of the best sequence* (Cazenave 2009) cannot be adapted to the two-player setting because of the alternation between maximizing and minimizing steps. Still, nothing prevents attempting to generalize some of the other heuristics such as the All-Moves-As-First idea (Akiyama, Komiya, and Kotani 2012) and the Nested Rollout Policy Adaptation (Rosin 2011) in future work. Future work could also examine how to further generalize NMCS to multi-outcome games.

While we built our Nested Player around a purely random policy as is most common in the GGP community (Björnsson and Finnsson 2009; Méhat and Cazenave 2010; Genesereth and Thielscher 2014), our technique could also build on the alternative domain-specific pseudo-random policies developed in the Computer Go community (Silver and Tesauro 2009; Browne et al. 2012). The interplay between such smart elementary playouts and our nesting construction and heuristics could provide a fruitful avenue for an experimentally oriented study.

Acknowledgments

This research was supported by the Australian Research Council under grant no. DP120102023 and DE150101351. The last author is also affiliated with the University of Western Sydney.

This work was granted access to the HPC resources of MesoPSL financed by the Region Ile de France and the project Equip@Meso (reference ANR-10-EQPX-29-01) of the programme Investissements d'Avenir supervised by the Agence Nationale pour la Recherche.

References

Akiyama, H.; Komiya, K.; and Kotani, Y. 2012. Nested Monte-Carlo search with simulation reduction. *Knowledge-Based Systems* 34:12–20.

Baier, H., and Winands, M. H. 2013. Monte-Carlo tree search and minimax hybrids. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 1–8. Niagara Falls, Canada: IEEE Press.

Björnsson, Y., and Finnsson, H. 2009. CADIAPLAYER: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games* 1(1):4–15.

Browne, C.; Powley, E.; Whitehouse, D.; Lucas, S.; Cowling, P.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Cazenave, T., and Saffidine, A. 2010. Score bounded Monte-Carlo tree search. In *7th International Conference on Computers and Games (CG)*, volume 6515 of *Lecture Notes in Computer Science*, 93–104. Kanazawa, Japan: Springer.

Cazenave, T. 2009. Nested Monte-Carlo search. In *21st International Joint Conference on Artificial Intelligence (IJCAI)*, 456–461. Pasadena, California, USA: AAAI Press.

Cazenave, T. 2010. Nested Monte-Carlo expression discovery. In *19th European Conference on Artificial Intelligence (ECAI)*, 1057–1058. Lisbon, Portugal: IOS Press.

Chaslot, G. M.-B.; Hoock, J.-B.; Perez, J.; Rimmel, A.; Teytaud, O.; and Winands, M. H. 2009. Meta Monte-Carlo Tree Search for automatic opening book generation. In *1st International General Game Playing Workshop (GIGA)*, 7–12.

Ewalds, T. 2012. Playing and solving Havannah. Master's thesis, University of Alberta.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI*, volume 8, 259–264.

Furtak, T., and Buro, M. 2013. Recursive Monte Carlo search for imperfect information games. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 225–232. Niagara Falls, Canada: IEEE Press.

Genesereth, M., and Thielscher, M. 2014. *General Game Playing*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool.

Genesereth, M.; Love, N.; and Pell, B. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2):62–72.

Gudmundsson, S. F., and Björnsson, Y. 2013. Sufficiency-based selection strategy for MCTS. In *23rd International Joint Conference on Artificial Intelligence (IJCAI)*, 559–565. Beijing, China: AAAI Press.

Knuth, D. E., and Moore, R. W. 1975. An analysis of alpha-beta pruning. *Artificial Intelligence* 6(4):293–326.

Lorentz, R. J. 2010. Improving Monte–Carlo tree search in havannah. In *7th International Conference on Computers and Games (CG)*, volume 6515 of *Lecture Notes in Computer Science*. Kanazawa, Japan: Springer. 105–115.

Méhat, J., and Cazenave, T. 2010. Combining UCT and nested Monte-Carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games* 2(4):271–277.

Pepels, T.; Tak, M. J.; Lanctot, M.; and Winands, M. H. 2014. Quality-based rewards for Monte-Carlo Tree Search simulations. In *21st European Conference on Artificial Intelligence (ECAI)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 705–710. Prague, Czech Republic: IOS Press.

Rosin, C. D. 2011. Nested rollout policy adaptation for Monte Carlo tree search. In *22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 649–654. Barcelona, Catalonia, Spain: AAAI Press.

Saffidine, A.; Jouandeau, N.; and Cazenave, T. 2011. Solving Breakthough with race patterns and Job-Level Proof Number Search. In *13th International Conference on Advances in Computer Games (ACG)*, volume 7168 of *Lecture Notes in Computer Science*, 196–207. Tilburg, The Netherlands: Springer.

Silver, D., and Tesauro, G. 2009. Monte-Carlo simulation balancing. In *26th International Conference on Machine Learning (ICML)*, 945–952. Montreal, Quebec, Canada: ACM.

Smith, S. J., and Nau, D. S. 1994. An analysis of forward pruning. In *12th National Conference on Artificial Intelligence (AAAI)*, 1386–1391. Seattle, WA, USA: AAAI Press.

Steinhauer, J. 2010. Monte-Carlo twixt. Master's thesis, Maastricht University, Maastricht, The Netherlands.

Winands, M. H., and Björnsson, Y. 2011. $\alpha\beta$-based play-outs in Monte-Carlo Tree Search. In *IEEE Conference on Computational Intelligence and Games (CIG)*, 110–117. Seoul, South Korea: IEEE Press.