Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence

Monte Carlo *-Minimax Search

Marc Lanctot
Department of Knowledge Engineering
Maastricht University, Netherlands
marc.lanctot@maastrichtuniversity.nl

Abdallah Saffidine
LAMSADE,
Université Paris-Dauphine, France
abdallah.saffidine@dauphine.fr

Joel Veness and **Christopher Archibald,**
Department of Computing Science
University of Alberta, Canada
{veness@cs., archibal}@ualberta.ca

Mark H. M. Winands
Department of Knowledge Engineering
Maastricht University, Netherlands
m.winands@maastrichtuniversity.nl

Abstract

This paper introduces Monte Carlo *-Minimax Search (MCMS), a Monte Carlo search algorithm for turned-based, stochastic, two-player, zero-sum games of perfect information. The algorithm is designed for the class of densely stochastic games; that is, games where one would rarely expect to sample the same successor state multiple times at any particular chance node. Our approach combines sparse sampling techniques from MDP planning with classic pruning techniques developed for adversarial expectimax planning. We compare and contrast our algorithm to the traditional *-Minimax approaches, as well as MCTS enhanced with the Double Progressive Widening, on four games: Pig, EinStein Würfelt Nicht!, Can't Stop, and Ra. Our results show that MCMS can be competitive with enhanced MCTS variants in some domains, while consistently outperforming the equivalent classic approaches given the same amount of thinking time.

1 Introduction

Monte Carlo sampling has recently become a popular technique for online planning in large sequential games. For example UCT and, more generally, Monte Carlo Tree Search (MCTS) [Kocsis and Szepesvári, 2006; Coulom, 2007b] has led to an increase in the performance of Computer Go players [Lee *et al.*, 2009], and numerous extensions and applications have since followed [Browne *et al.*, 2012]. Initially, MCTS was applied to games lacking strong Minimax players, but recently has been shown to compete against strong Minimax players in such games [Winands *et al.*, 2010; Ramanujan and Selman, 2011]. One class of games that has proven more resistant is stochastic games. Unlike classic games such as Chess and Go, stochastic game trees include chance nodes in addition to decision nodes. How MCTS should account for this added uncertainty remains unclear. Moreover, many of the search enhancements from the classic $\alpha\beta$ literature cannot be easily adapted to MCTS. The classic algorithms for stochastic games, EXPECTIMAX and *-Minimax (Star1 and Star2), perform look-ahead searches to a limited depth. However, the running time of these algorithms scales exponentially in the branching factor at chance nodes as the search horizon is increased. Hence, their performance in large games often depends heavily on the quality of the heuristic evaluation function, as only shallow searches are possible.

One way to handle the uncertainty at chance nodes would be forward pruning [Smith and Nau, 1993], but the performance gain until now has been small [Schadd *et al.*, 2009]. Another way is to simply sample a single outcome when encountering a chance node. This is common practice in MCTS when applied to stochastic games. However, the general performance of this method is unknown. Large stochastic domains still pose a significant challenge. For instance, MCTS is outperformed by *-Minimax in the game of Carcassonne [Heyden, 2009]. Unfortunately, the literature on the application of Monte Carlo search methods to stochastic games is relatively small.

In this paper, we investigate the use of Monte Carlo sampling in *-Minimax search. We introduce a new algorithm, Monte Carlo *-Minimax Search (MCMS), which samples a subset of chance node outcomes in EXPECTIMAX and *-Minimax in stochastic games. In particular, we describe a sampling technique for chance nodes based on sparse sampling [Kearns *et al.*, 1999] and show that MCMS approaches the optimal decision as the number of samples grows. We evaluate the practical performance of MCMS in four domains: Pig, EinStein Würfelt Nicht!, Can't Stop, and Ra. In Pig, we show that the estimates returned by MCMS have lower bias and lower regret than the estimates returned by the classic *-Minimax algorithms. Finally, we show that the addition of sampling to *-Minimax can increase its performance from inferior to competitive against state-of-the-art MCTS, and in the case of Ra, can even perform better than MCTS.

2 Background

A finite, two-player zero-sum game of perfect information can be described as a tuple $(\mathcal{S}, \mathcal{T}, \mathcal{A}, \mathcal{P}, u_1, s_1)$, which we now define. The state space $\mathcal{S}$ is a finite, non-empty set of states, with $\mathcal{T} \subseteq \mathcal{S}$ denoting the finite, non-empty set of terminal states. The action space $\mathcal{A}$ is a finite, non-empty set of actions. The transition probability function $\mathcal{P}$ assigns to each state-action pair $(s,a) \in \mathcal{S} \times \mathcal{A}$ a probability measure over $\mathcal{S}$ that we denote by $\mathcal{P}(\cdot \mid s,a)$. The utility function $u_1 : \mathcal{T} \mapsto [v_{\min}, v_{\max}] \subseteq \mathbb{R}$ gives the utility of player 1, with v_{min} and $v_{\max}$ denoting the minimum and maximum possible utility, respectively. Since the game is zero-sum, the utility of player 2 in any state $s \in \mathcal{T}$ is given by $u_2(s) := -u_1(s)$. The player index function $\tau : \mathcal{S} \setminus \mathcal{T} \to \{1,2\}$ returns the player to act in a given non-terminal state s.

Each game starts in the initial state s_1 with $\tau(s_1) := 1$, and proceeds as follows. For each time step $t \in \mathbb{N}$, player $\tau(s_t)$ selects an action $a_t \in \mathcal{A}$ in state s_t, with the next state s_{t+1} generated according to $\mathcal{P}(\cdot \mid s_t, a_t)$. Player $\tau(s_{t+1})$ then chooses a next action and the cycle continues until some terminal state $s_T \in \mathcal{T}$ is reached. At this point player 1 and player 2 receive a utility of $u_1(s_T)$ and $u_2(s_T)$ respectively.

2.1 Classic Game Tree Search

We now describe the two main search paradigms for adversarial stochastic game tree search. We begin by first describing classic stochastic search techniques, that differ from modern approaches in that they do not use Monte Carlo sampling. This requires recursively defining the minimax value of a state $s \in \mathcal{S}$, which is given by

$$V(s) = \begin{cases} \max\limits_{a \in \mathcal{A}} \sum\limits_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a)\, V(s') & \text{if } s \notin \mathcal{T}, \tau(s) = 1 \\ \min\limits_{a \in \mathcal{A}} \sum\limits_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a)\, V(s') & \text{if } s \notin \mathcal{T}, \tau(s) = 2 \\ u_1(s) & \text{otherwise.} \end{cases}$$

Note that here we always treat player 1 as the player maximizing $u_1(s)$ (*Max*), and player 2 as the player minimizing $u_1(s)$ (*Min*). In most large games, computing the minimax value for a given game state is intractable. Because of this, an often used approximation is to instead compute the *depth d minimax value*. This requires limiting the recursion to some fixed depth $d \in \mathbb{N}$ and applying a heuristic evaluation function when this depth limit is reached. Thus given a heuristic evaluation function $h : \mathcal{S} \to [v_{\min}, v_{\max}] \subseteq \mathbb{R}$ defined with respect to player 1 that satisfies the requirement $h(s) = u_1(s)$ when $s \in \mathcal{T}$, the depth d minimax value is defined recursively by

$$V_d(s) = \begin{cases} \max\limits_{a \in \mathcal{A}} V_d(s,a) & \text{if } d > 0,\ s \notin \mathcal{T}, \text{ and } \tau(s) = 1 \\ \min\limits_{a \in \mathcal{A}} V_d(s,a) & \text{if } d > 0,\ s \notin \mathcal{T}, \text{ and } \tau(s) = 2 \\ h(s) & \text{otherwise,} \end{cases}$$

where

$$V_d(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}(s' \mid s,a)\, V_{d-1}(s'). \tag{1}$$

For sufficiently large d, $V_d(s)$ coincides with $V(s)$. The quality of the approximation depends on both the heuristic evaluation function and the search depth parameter d.

A direct computation of $\arg\max_{a \in \mathcal{A}(s)} V_d(s,a)$ or $\arg\min_{a \in \mathcal{A}(s)} V_d(s,a)$ is equivalent to running the well known EXPECTIMAX algorithm [Michie, 1966]. The base EXPECTIMAX algorithm can be enhanced by a technique similar to $\alpha\beta$ pruning [Knuth and Moore, 1975] for deterministic game tree search. This involves correctly propagating the $[\alpha, \beta]$ bounds and performing an additional pruning step at each chance node. This pruning step is based on the observation that if the minimax value has already been computed for a subset of successors $\tilde{\mathcal{S}} \subseteq \mathcal{S}$, the depth d minimax value of state-action pair (s,a) must lie within

$$L_d(s,a) \leq V_d(s,a) \leq U_d(s,a),$$

where

$$L_d(s,a) = \sum_{s' \in \tilde{\mathcal{S}}} \mathcal{P}(s' \mid s,a) V_{d-1}(s') + \sum_{s' \in \mathcal{S} \setminus \tilde{\mathcal{S}}} \mathcal{P}(s' \mid s,a) v_{\min}$$

$$U_d(s,a) = \sum_{s' \in \tilde{\mathcal{S}}} \mathcal{P}(s' \mid s,a) V_{d-1}(s') + \sum_{s' \in \mathcal{S} \setminus \tilde{\mathcal{S}}} \mathcal{P}(s' \mid s,a) v_{\max}.$$

These bounds form the basis of the pruning mechanisms in the *-Minimax [Ballard, 1983] family of algorithms. In the Star1 algorithm, each s' from the equations above represents the state reached after a particular outcome is applied at a chance node following (s,a). In practice, Star1 maintains lower and upper bounds on $V_{d-1}(s')$ for each child s' at chance nodes, using this information to stop the search when it finds a proof that any future search is pointless. A worked example of how these cuts occur in *-Minimax can be found in [Lanctot *et al.*, 2013].

```
1  Star1(s, a, d, α, β)
2      if d = 0 or s ∈ T then return h(s)
3
4      else
5          O ← genOutcomeSet(s, a)
6          for o ∈ O do
7              α' ← childAlpha(o, α)
8              β' ← childBeta(o, β)
9              s' ← actionChanceEvent(s, a, o)
10             v ← alphabeta1(s', d − 1, α', β')
11             o_l ← v; o_u ← v
12             if v ≥ β' then return pess(O)
13
14             if v ≤ α' then return opti(O)
15
16         return V_d(s, a)
```

Algorithm 1: Star1

The algorithm is summarized in Algorithm 1. The `alphabeta1` procedure recursively calls Star1. The outcome set $\mathcal{O}$ is an array of tuples, one per outcome. One such tuple o has three attributes: a lower bound o_l initialized to $v_{\min}$, an upper bound o_u initialized to $v_{\max}$, and the outcome's probability o_p. The pess function returns the current lower bound on the chance node $\text{pess}(\mathcal{O}) = \sum_{o \in \mathcal{O}} o_p o_l$. Similarly, opti returns the current upper bound on the chance

node using o_u in place of o_l: $\text{opti}(\mathcal{O}) = \sum_{o \in \mathcal{O}} o_p o_u$. Finally, the functions `childAlpha` and `childBeta` return the new bounds on the value of the respective child below. In general:

$$\alpha' = \max\left\{v_{\min}, \frac{\alpha - \text{opti}(\mathcal{O}) + o_p o_u}{o_p}\right\},$$

$$\beta' = \min\left\{v_{\max}, \frac{\beta - \text{pess}(\mathcal{O}) + o_p o_l}{o_p}\right\}.$$

The performance of the algorithm can be improved significantly by applying a simple look-ahead heuristic. Suppose the algorithm encounters a chance node. When searching the children of each outcome, one can temporarily restrict the legal actions at a successor (decision) node. If only a single action is searched at the successor, then the value returned will be a bound on $V_{d-1}(s')$. If the successor is a Max node, then the true value can only be larger, and hence the value returned is a lower bound. Similarly, if it was a Min node, the value returned is a lower bound. The Star2 algorithm applies this idea via a preliminary *probing phase* at chance nodes in hopes of pruning without requiring full search of the children. If probing does not lead to a cutoff, then the children are fully searched, but bound information collected in the probing phase can be re-used. When moves are appropriately ordered, the algorithm can often choose the best single move and effectively cause a cut-off with much less search effort. Since this idea is applied recursively, the benefits compounds as the depth increases. The algorithm is summarized in Algorithm 2. The `alphabeta2` procedure is analogous to `alphabeta1` except when p is true, a subset (of size one) of the actions are considered at the next decision node. The recursive calls to Star2 within `alphabeta2` have p set to false and a set to the chosen action.

Star1 and Star2 are typically presented using the negamax formulation. In fact, Ballard originally restricted his discussion to regular *-Minimax trees, ones that strictly alternate Max, Chance, Min, Chance. We intentionally present the more general $\alpha\beta$ formulation here because it handles a specific case encountered by three of our test domains. In games where the outcome of a chance node determines the next player to play, the cut criteria during the Star2 probing phase depends on the child node. The bound established by the Star2 probing phase will either be a lower bound or an upper bound, depending on the child's type. This distinction is made in lines 11 to 17. Also note: when implementing the algorithm, for better performance it is advisable to incrementally compute the bound information [Hauk *et al.*, 2006].

```
1  Star2(s, a, d, α, β)
2      if d = 0 or s ∈ T then return h(s)
3
4      else
5          O ← genOutcomeSet(s, a)
6          for o ∈ O do
7              α' ← childAlpha(o, α)
8              β' ← childBeta(o, β)
9              s' ← actionChanceEvent(s, a, o)
10             v ← alphabeta2(s', d − 1, α', β', true)
11             if τ(s') = 1 then
12                 o_l ← v
13                 if pess(O) ≥ β then return pess(O)
14
15             else if τ(s') = 2 then
16                 o_u ← v
17                 if opti(O) ≤ α then return opti(O)
18
19         for o ∈ O do
20             α' ← childAlpha(o, α)
21             β' ← childBeta(o, β)
22             s' ← actionChanceEvent(s, a, o)
23             v ← alphabeta2(s', d − 1, α', β', false)
24             o_l ← v; o_u ← v
25             if v ≥ β' then return pess(O)
26
27             if v ≤ α' then return opti(O)
28
29         return V_d(s, a)
```

Algorithm 2: Star2

2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) has attracted significant attention in recent years. The main idea is to iteratively run simulations from the game's current position to a leaf, incrementally growing a tree rooted at the current position. In its simplest form, the tree is initially empty, with each simulation expanding the tree by an additional node. When this node is not terminal, a rollout policy takes over and chooses actions until a terminal state is reached. Upon reaching a terminal state, the observed utility is back-propagated through all the nodes visited in this simulation, which causes the value estimates to become more accurate over time. This idea of using random rollouts to estimate the value of individual positions has proven successful in Go and many other domains [Coulom, 2007b; Browne *et al.*, 2012].

While descending through the tree, a sequence of actions must be selected for further exploration. A popular way to do this so as to balance between exploration and exploitation is to use algorithms developed for the well-known stochastic multi-armed bandit problem [Auer *et al.*, 2002]. UCT is an algorithm that recursively applies one of these selection mechanism to trees [Kocsis and Szepesvári, 2006]. An improvement of significant practical importance is progressive unpruning / widening [Coulom, 2007a; Chaslot *et al.*, 2008]. The main idea is to purposely restrict the number of allowed actions, with this restriction being slowly relaxed so that the tree grows deeper at first and then slowly wider over time. Progressive widening has also been extended to include chance nodes, leading to the Double Progressive Widening algorithm (DPW) [Couetoux *et al.*, 2011]. When DPW encounters a chance or decision node, it computes a maximum number of actions or outcomes to consider $k = \lceil C v^{\alpha} \rceil$, where C and α are parameter constants and v represents a number of visits to the node. At a decision node, then only the first k actions from the action set are available. At a chance node, a set of outcomes is stored and incrementally grown. An outcome is sampled; if k is larger than the size of the current set

of outcomes and the newly sampled outcome is not in the set, it is added to the set. Otherwise, DPW samples from existing children at chance nodes in the tree, where a child's probability is computed with respect to the current children in the restricted set. This enhancement has been shown to improve the performance of MCTS in densely stochastic games.

2.3 Sampling in Markov Decision Processes

Computing optimal policies in large Markov Decision Processes (MDPs) is a significant challenge. Since the size of the state space is often exponential in the properties describing each state, much work has focused on finding efficient methods to compute approximately optimal solutions. One way to do this, given only a generative model of the domain, is to employ *sparse sampling* [Kearns *et al.*, 1999]. When faced with a decision to make from a particular state, a local sub-MDP can be built using fixed depth search. When transitioning to successor states, a fixed number $c \in \mathbb{N}$ of successor states are sampled for each action. Kearns et al. showed that for an appropriate choice of c, this procedure produces value estimates that are accurate with high probability. Importantly, c was shown to have no dependence on the number of states $|\mathcal{S}|$, effectively breaking the curse of dimensionality. This method of sparse sampling was later improved by using adaptive decision rules based on the multi-armed bandit literature to give the AMS algorithm [Chang *et al.*, 2005]. Also, the Forward Search Sparse Sampling (FSSS) [Walsh *et al.*, 2010] algorithm was recently introduced, which exploits bound information to add a form of sound pruning to sparse sampling. The branch and bound pruning mechanism used by FSSS works similarly to Star1 in adversarial domains.

3 Sparse Sampling in Adversarial Games

The practical performance of classic game tree search algorithms such as Star1 or Star2 strongly depend on the typical branching factor at chance nodes. Since this can be as bad as $|\mathcal{S}|$, long-term planning using classic techniques is often infeasible in stochastic domains. However, like sparse sampling for MDPs in Section 2.3, this dependency can be removed by an appropriate use of Monte Carlo sampling. We now define the *estimated depth* d *minimax value* as

$$\hat{V}_d(s) := \begin{cases} \max_{a \in \mathcal{A}} \hat{V}_d(s,a) & \text{if } d > 0, s \notin \mathcal{T}, \text{ and } \tau(s) = 1 \\ \min_{a \in \mathcal{A}} \hat{V}_d(s,a) & \text{if } d > 0, s \notin \mathcal{T}, \text{ and } \tau(s) = 2 \\ h(s) & \text{otherwise.} \end{cases}$$

where

$$\hat{V}_d(s,a) := \tfrac{1}{c} \sum_{i=1}^{c} \hat{V}_{d-1}(s_i),$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, with each successor state s_i distributed according to $\mathcal{P}(\cdot \,|\, s, a)$ for $1 \leq i \leq c$. This natural definition can be justified by the following result, which shows that the value estimates are accurate with high probability, provided c is chosen to be sufficiently large.

Theorem 1. *Given* $c \in \mathbb{N}$, *for any state* $s \in \mathcal{S}$, *for all* $\lambda \in (0, 2v_{\max}] \subset \mathbb{R}$, *for any depth* $d \in \mathbb{Z}_+$,

$$\mathbb{P}\left(\left|\hat{V}_d(s) - V_d(s)\right| \leq \lambda d\right) \geq 1 - (2c|\mathcal{A}|)^d \exp\left\{\frac{-\lambda^2 c}{2v_{\max}^2}\right\}.$$

The proof is a straightforward generalization of the result of Kearns *et al.* [1999] for finite horizon, adversarial games and can be found in [Lanctot *et al.*, 2013].

Notice that although there is no dependence on $|\mathcal{S}|$, there is still an exponential dependence on the horizon d. Thus an enormously large value of c will need to be used to obtain any meaningful theoretical guarantees. Nevertheless, we shall show later that surprisingly small values of c perform well in practice. Also note that our proof of Theorem 1 does not hold when sampling without replacement is used. Investigating whether the analysis can be extended to cover this case would be an interesting next step.

3.1 Monte Carlo *-Minimax

We are now in a position to describe the MCMS family of algorithms, which compute estimated depth d minimax values by recursively applying one of the Star1 or Star2 pruning rules. The MCMS variants can be easily described in terms of the previous descriptions of the original Star1 and Star2 algorithms. To enable sampling, one need only change the implementation of `getOutcomeSet` on line 5 of Algorithm 1 and line 5 of Algorithm 2. At a chance node, instead of recursively visiting the subtrees under each outcome, c outcomes are sampled *with replacement* and only the subtrees under those outcomes are visited; the value returned to the parent is the (equally weighted) average of the c samples. Equivalently, one can view this approach as transforming each chance node into a new chance node with c outcomes, each having probability $\frac{1}{c}$. We call these new variants star1SS and star2SS. If all pruning is disabled, we obtain EXPECTIMAX with sparse sampling (expSS), which computes $\hat{V}_d(s)$ directly from definition. At a fixed depth, if both algorithms sample identically the star1SS method computes exactly the same value as expSS but will avoid useless work by using the Star1 pruning rule. The case of star2SS is slightly more complicated. For Theorem 1 to apply, the bound information collected in the probing phase needs to be consistent with the bound information used after the probing phase. To ensure this, the algorithm must sample outcomes identically in the subtrees taken while probing and afterward.

4 Empirical Evaluation

We now describe our experiments. We start with our domains: Pig, EinStein Würfelt Nicht!, Can't Stop, and Ra. We then describe in detail our experiment setup. We then describe two experiments: one to determine the individual performance of each algorithm, and one to compute the statistical properties of the underlying estimators.

4.1 Domains

Pig is a two-player dice game [Scarne, 1945]. Players each start with 0 points; the goal is to be the first player to achieve 100 or more points. Each turn, players roll two dice and then,

if there are no ⚀ showing, add the sum to their turn total. At each decision point, a player may continue to roll or stop. If they decide to stop, they add their turn total to their total score and then it becomes the opponent's turn. Otherwise, they roll dice again for a chance to continue adding to their turn total. If a single ⚀ is rolled the turn total will be reset and the turn ended (no points gained); if a ⚀⚀ is rolled then the players turn will end along with their *total score* being reset to 0.

EinStein Würfelt Nicht! (EWN) is a game played on a 5 by 5 square board. Players start with six dice used as pieces (⚀, ⚁, ..., ⚅) in opposing corners of the board. The goal is to reach the opponent's corner square with a single die or capture every opponent piece. Each turn starts with the player rolling a neutral six-sided die whose result indicates which one of their pieces (dice) can move this turn. Then the player must move a piece toward the opponent's corner base (or off the board). Whenever moving onto a square with a piece, it is captured. EWN is a game played by humans and computer opponents on the Little Golem online board game site; at least two MCTS players have been developed to play it [Lorentz, 2011; Shahbazian, 2012].

Can't Stop is a dice game [Sackson, 1980] that is very popular on online gaming sites.[1] Can't Stop has also been a domain of interest to AI researchers [Glenn and Aloi, 2009; Fang *et al.*, 2008]. The goal is to obtain three complete columns by reaching the highest level in each of the 2-12 columns. This is done by repeatedly rolling 4 dice and playing zero or more pairing combinations. Once a pairing combination is played, a marker is placed on the associated column and moved upwards. Only three distinct columns can be used during any given turn. If dice are rolled and no legal pairing combination can be made, the player loses all of the progress made towards completing columns on this turn. After rolling and making a legal pairing, a player can chose to lock in their progress by ending their turn.

Ra is a set collection bidding game, currently ranked #58 highest board game (out of several thousand) on the community site BoardGameGeek.com. Players collect various combinations of tiles by winning auctions using the bidding tokens (*suns*). Each turn, a player chooses to either draw a tile from the bag or start an auction. When a special Ra tile is drawn, an auction starts immediately, and players use one of their suns to bid on the current group of tiles. By winning an auction, a player takes the current set of tiles and exchanges the winning sun with the one in the middle of the board, the one gained becoming inactive until the following round (*epoch*). When a player no longer has any active suns, they cannot take their turns until the next epoch. Points are attributed to each player at the end of each epoch depending on their tile set as well as the tile sets of other players.

4.2 Experimental Setup

In our implementation, low-overhead static move orderings are used to enumerate actions. Iterative deepening is used so that when a timeout occurs, if a move at the root has not been fully searched, then the best move from the previous depth search is returned. Transposition tables are used to store the

[1] See the yucata.de and boardgamearena.com statistics.

Table 1: Mean statistical property values over 2470 Pig states.

Algorithm	Property			
	MSE	Variance	|Bias|	Regret
MCTS	78.7	0.71	8.83	0.41
DPW	79.4	5.3	8.61	0.96
exp	91.4	0.037	9.56	0.56
Star1	91.0	0.064	9.54	0.55
Star2	87.9	0.008	9.38	0.58
expSS	95.3	13.0	9.07	0.52
star1SS	99.8	11.0	9.43	0.55
star2SS	97.5	14.8	9.09	0.56

best move to improve move ordering for future searches. In addition, to account for the extra overhead of maintaining bound information, pruning is ignored at search depths 2 or lower. In MCTS, chance nodes are stored in the tree and the selection policy always samples an outcome based on their probability distributions, which are non-uniform in every case except EWN.

Our experiments use a search time limit of 200 milliseconds. MCTS uses utilities in $[-100, 100]$ and a UCT exploration constant of C_1. Since evaluation functions are available, we augment MCTS with a parameter, d_r, representing the number of moves taken by the rollout policy before the evaluation function is called. MCTS with double-progressive widening (DPW) uses two more parameters C_2 and α described in Section 2.2. Each algorithm's parameters are tuned via self-play tournaments where each player in the tournament represents a specific parameter set from a range of possible parameters and seats are swapped to ensure fairness. Specifically we used a multi-round elimination style tournament where head-to-head pairing consisted of 1000 games (500 swapped seat matches) between two different sets of parameters, winners continuing to the next round, and final champion determining the optimal parameter values. By repeating the tournaments, we found this elimination style tuning to be more consistent than round-robin style tournament, even with a larger total number of games. The sample widths for (expSS, star1SS, star2SS) in Pig were found to be $(20, 25, 18)$. In EWN, Can't Stop, and Ra, they were found to be $(1, 1, 2)$, $(25, 30, 15)$, and $(5, 5, 2)$ respectively. In MCTS and DPW, the optimal parameters (C_1, d_r, C_2, α) in Pig were found to be $(50, 0, 5, 0.2)$. In EWN, Can't Stop, and Ra, they were found to be $(200, 100, 4, 0.25)$, $(50, 10, 25, 0.3)$, and $(50, 0, 2, 0.1)$ respectively. The values of d_r imply that the quality of the evaluation function in EWN is significantly lower than in other games.

4.3 Statistical Properties

Our first experiment compares statistical properties of the estimates and actions returned by *-Minimax, MCMS, and MCTS. At a single decision point s, each algorithm acts as an estimator of the true minimax value $\hat{V}(s)$, and returns the action $a \in \mathcal{A}$ that maximizes $\hat{V}(s, a)$. Since Pig has fewer than one million states, we solve it using the technique of

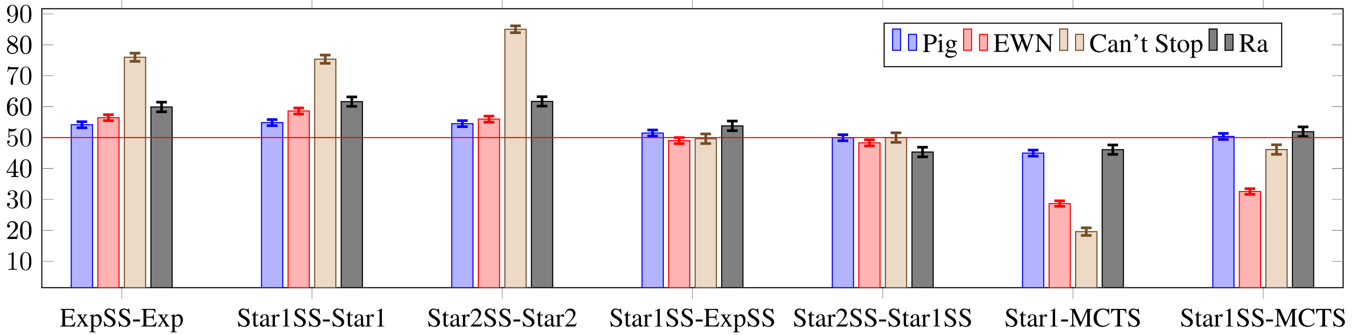

Figure 1: Results of playing strength experiments. Each bar represents the percentage of wins for p_{left} in a p_{left}-p_{right} pairing. (Positions are swapped and this notation refers only to the name order.) Errors bars represent 95% confidence intervals. Here, the best variant of MCTS is used in each domain. exp-MCTS, expSS-MCTS, Star2-MCTS, and star2SS-expSS are intentionally omitted since they look similar to Star1-MCTS, star1SS-MCTS, Star1-MCTS, and star1SS-expSS, respectively.

value iteration which has been applied to previous smaller games of Pig [Neller and Pressor, 2004], obtaining the true value of each state $V(s)$. From this, we estimate the *mean squared error*, *variance*, *bias*, and *regret* of each algorithm using $\text{MSE}[\hat{V}(s)] = \mathbb{E}[(\hat{V}(s) - V(s))^2] = \mathbb{V}\text{ar}[\hat{V}(s)] + \text{Bias}(V(s), \hat{V}(s))^2$ by running each algorithm 50 separate times at each decision point. Then we compute the regret of taking action a at state s, $\text{Regret}(s, a) = V(s) - V(s, a)$, where a is the action chosen by the algorithm from state s. As with MSE, variance, and bias: for a state s, we estimate $\text{Regret}(s, a)$ by computing a mean over 50 runs starting at s. The estimates of these properties are computed for each state in a collection of states $s \in \mathcal{S}_{obs}$ observed through simulated games. $\mathcal{S}_{obs}$ is formed by taking every state seen through simulated games of each type of player plays against each other type of player, and discarding duplicate states. Therefore, the states collected represent states that actually visited during game play. We then report the average value of each property over these $|\mathcal{S}_{obs}| = 2470$ game states are shown in Table 1.

The results in the table show the trade-offs between bias and variance. We see that the estimated bias returned by expSS are lower than the classic *-Minimax algorithms. The performance results below may be explained by this reduction in bias. While variance is introduced due to sampling, seemingly causing higher MSE, in two of three cases the regret in MCMS is lower than *-Minimax which ultimately leading to better performance, as seen in the following section.

4.4 Playing Strength

In our second experiment, we computed the performance of each algorithm by playing a number of test matches (5000 for Pig and EWN, 2000 for Can't Stop and Ra) for each paired set of players. Each *match* consists of two games where players swap seats and a single randomly generated seed is used for both games in the match. To determine the best MCTS variant, 500 matched of MCTS versus DPW were played in each domain, and the winner was chosen; (classic MCTS in Pig and EWN, DPW Can't Stop and Ra). The performance of each pairing of players is shown in Figure 1.

The results show that the MCMS variants outperform their equivalent classic counterparts in every case, establishing a clear benefit of sparse sampling in the *-Minimax algorithm. In some cases, the improvement is quite significant, such as an 85.0% win rate for star2SS vs Star2 in Can't Stop. MCMS also performs particularly well in Ra obtaining roughly 60% wins against it classic *-Minimax counterparts. This indicates that MCMS is well suited for densely stochastic games. In Pig and Ra, the best MCMS variant seems to perform comparably to the best variant of MCTS; the weak performance of EWN is likely due to the lack of a good evaluation function. Nonetheless, when looking at the relative performances of classic *-Minimax, we see the performance against MCTS improves when sparse sampling is applied. We also notice that in EWN expSS slightly outperforms star1SS; this can occur when there are few pruning opportunities and the overhead added by maintaining the bound information outweighs the benefit of pruning. A similar phenomenon is observed for star1SS and star2SS in Ra.

The relative performance between expSS, star1SS, and star2SS is less clear. This could be due to the overhead incurred by maintaining bound information reducing the time saved by sampling; *i.e.* the benefit of additional sampling may be greater than the benefit of pruning within the smaller sample. We believe that the relative performance of the MCMS could improve with the addition of domain knowledge such as classic search heuristics and specially tuned evaluation functions that lead to more pruning opportunities, but more work is required to show this.

5 Conclusion and Future Work

This paper has introduced MCMS, a family of sparse sampling algorithms for two-player, perfect information, stochastic, adversarial games. Our results show that MCMS can be competitive against MCTS variants in some domains, while consistently outperforming the equivalent classic approaches given the same amount of thinking time. We feel that our initial results are encouraging, and worthy of further investigation. One particularly attractive property of MCMS compared with MCTS (and variants) is the ease in which other classic pruning techniques can be incorporated. This could lead to larger per-

formance improvements in domains where forward pruning techniques such as Null-move Pruning or Multicut Pruning are known to work well.

For future work, we plan to investigate the effect of dynamically set sample widths, sampling without replacement, and the effect of different time limits. In addition, as the sampling introduces variance, the variance reduction techniques used in MCTS [Veness *et al.*, 2011] may help in improving the accuracy of the estimates. Finally, we would like to determine the playing strength of MCMS algorithms against known AI techniques for these games [Glenn and Aloi, 2009; Fang *et al.*, 2008].

Acknowledgments

This work is partially funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project Go4Nature, grant number 612.000.938.

References

P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

B.W. Ballard. The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

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

H.S. Chang, M.C. Fu, J. Hu, and S.I. Marcus. An adaptive sampling algorithm for solving markov decision processes. *Operations Research*, 53(1):126–139, January 2005.

G.M.J-B. Chaslot, M.H.M. Winands, H.J. van den Herik, J.W.H.M. Uiterwijk, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.

A. Couetoux, J-B. Hoock, N. Sokolovska, O. Teytaud, and N. Bonnard. Continuous upper confidence trees. In *LION'11: Proceedings of the 5th International Conference on Learning and Intelligent OptimizatioN*, Italy, 2011.

R. Coulom. Computing "ELO ratings" of move patterns in the game of Go. *ICGA Journal*, 30(4):198–208, 2007.

R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Proceedings of the 5th international conference on Computers and games*, CG'06, pages 72–83, Berlin, Heidelberg, 2007. Springer-Verlag.

H. Fang, J. Glenn, and C. Kruskal. Retrograde approximation algorithms for jeopardy stochastic games. *ICGA journal*, 31(2):77–96, 2008.

J. Glenn and C. Aloi. Optimizing genetic algorithm parameters for a stochastic game. In *Proceedings of 22nd FLAIRS Conference*, pages 421–426, 2009.

T. Hauk, M. Buro, and J. Schaeffer. Rediscovering *-minimax search. In *Proceedings of the 4th international conference on Computers and Games*, CG'04, pages 35–50, Berlin, Heidelberg, 2006. Springer-Verlag.

C. Heyden. Implementing a Computer Player for Carcassonnne. Master's thesis, Department of Knowledge Engineering, Maastricht University, 2009.

M.J. Kearns, Y. Mansour, and A.Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov Decision Processes. In *IJCAI*, pages 1324–1331, 1999.

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

L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, pages 282–293, 2006.

M. Lanctot, A. Saffidine, J. Veness, C. Archibald, and M.H.M. Winands. Monte carlo *-minimax search. *CoRR*, abs/1304.6057, 2013. http://arxiv.org/abs/1304.6057.

C-S. Lee, M-H. Wang, G. Chaslot, J-B. Hoock, A. Rimmel, O. Teytaud, S-R. Tsai, S-C. Hsu, and T-P. Hong. The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):73–89, 2009.

R.J. Lorentz. An MCTS program to Play EinStein Würfelt Nicht! In *Proceedings of the 12th International Conference on Advances in Computer Games*, pages 52–59, 2011.

D. Michie. Game-playing and game-learning automata. *Advances in Programming and Non-numerical Computation*, pages 183–196, 1966.

Todd W. Neller and Clifton G.M. Pressor. Optimal play of the dice game pig. *Undergraduate Mathematics and Its Applications*, 25(1):25–47, 2004.

R. Ramanujan and B. Selman. Trade-offs in sampling-based adversarial planning. In *ICAPS*, 2011.

S. Sackson. Can't Stop. *Ravensburger*, 1980.

J. Scarne. Scarne on Dice. *Harrisburg, PA: Military Service Publishing Co*, 1945.

M.P.D. Schadd, M.H.M. Winands, and J.W.H.M. Uiterwijk. ChanceProbcut: Forward pruning in chance nodes. In P.L. Lanzi, editor, *2009 IEEE Symposium on Computational Intelligence and Games (CIG'09)*, pages 178–185, 2009.

Sarmen Shahbazian. Monte Carlo tree search in EinStein Würfelt Nicht! Master's thesis, California State University, Northridge, 2012.

S.J.J. Smith and D.S. Nau. Toward an analysis of forward pruning. Technical Report CS-TR-3096, University of Maryland at College Park, College Park, 1993.

J. Veness, M. Lanctot, and M. Bowling. Variance reduction in Monte-Carlo Tree Search. In J. Shawe-Taylor, R.S. Zemel, P. Bartlett, F.C.N. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 1836–1844. 2011.

T.J. Walsh, S. Goschin, and M.L. Littman. Integrating sample-based planning and model-based reinforcement learning. In *AAAI*, 2010.

M.H.M. Winands, Y. Björnsson, and J-T. Saito. Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):239–250, 2010.