arXiv:2605.08982v3 [cs.LG] 7 Sep 2026

PMCTS: Principled Parallelized Inference Time Scaling with Particle Monte Carlo Tree Search

Yaniv Oren[1,5*] **Viliam Vadocz**[2*] **Joery A. de Vries**[3†] **Wendelin Böhmer**[1]

Matthijs T. J. Spaan[1] **Hendrik Baier**[4,5]

[1]Department of Intelligent Systems, TU Delft
[2]Department of Computer Science, ETH Zürich
[3]Trent AI Limited
[4]Information Systems, TU Eindhoven
[5]Centrum Wiskunde & Informatica, Amsterdam
{y.oren,j.w.bohmer,m.t.j.spaan}@tudelft.nl
vvadocz@student.ethz.ch
joery@trent.ai
h.j.s.baier@tue.nl

Abstract

Monte Carlo Tree Search (MCTS) is a widely used approach for policy improvement and action selection in Reinforcement Learning. Due to its sequential and deterministic nature, principled runtime-scaling of MCTS with parallel compute remains a major challenge. We introduce Particle MCTS (PMCTS), a principled parallel MCTS algorithm suited for neural network evaluations and designed for GPU-acceleration with batch-parallelization. We establish *policy improvement* properties for modern MCTS variants, and show that PMCTS maintains them. Empirically, PMCTS scales well with parallel compute and consistently outperforms or compares well to the popular heuristic-based baselines across a range of MCTS and RL evaluation domains, including the board games chess and Go and popular discrete action and continuous control benchmarks.

1 Introduction

Monte Carlo Tree Search [MCTS, 1] is a popular search algorithm that plays a central role in modern reinforcement learning (RL). MCTS performs *policy improvement* and action selection through lookahead and evaluation (i.e., *search*) in a model of the dynamics of the environment. This paradigm has been popularized in RL by the AlphaZero family of algorithms [2, 3] and has since been successfully applied to a wide range of domains, including zero-sum games [3], discrete-action, single-agent tasks [4–7] as well as continuous control [8]. The wide applicability of MCTS, including for practical applications [5–7] and LLM inference scaling [9–16], makes its inference-time scaling with parallel compute a critical challenge and the subject of significant attention [17–22].

Parallelizing MCTS is challenging for two main reasons: (I) In each iteration, MCTS selects a trajectory for evaluation using conventionally deterministic selection policies that depend on statistics from previous iterations in a deterministic dynamics model [1, 23, 24]. This makes the *principled*

*Equal contribution.
†Work done outside of Trent AI.

Preprint.

selection of multiple *unique* trajectories in parallel a significant challenge, because repeated parallel selection steps result in the same trajectory. (II) Modern MCTS methods use a deterministic deep neural network (DNN) for evaluation. This makes scaling the evaluation of a *single* trajectory with parallel compute a significant challenge, because repeated evaluations of the same trajectory result in the same evaluation rather than improved evaluation. Together, these properties also make independent instances of MCTS at the same state (*root parallelization*) expand similar trees and return similar search results, which prevents root parallelization from being an effective inference-time scaling method and makes the selection of multiple unique trajectories in parallel a critical challenge.

To overcome these challenges popular methods use heuristic selection policies such as *Virtual Losses* [18] and *Virtual Means* [22] which rely on communication between processes during selection. These methods suffer from three main limitations: first, they do not provide principled policy improvement. Their heuristic selection policy does not have a clear interpretation as an improved policy and thus no guarantees that the search results in policy improvement. Second, communication during selection hamstrings parallelization and precludes batched-parallelization, which is an effective and popular approach for GPU-acceleration in RL specifically and Deep Learning in general [25]. Third, they do not correctly account for the effect of duplicate trajectories on the evaluation.

To address these limitations, in this work we introduce *Particle MCTS* (PMCTS), a novel parallel MCTS algorithm which is principled, well suited for GPU acceleration through batch-parallelization and correctly accounts for the presence of duplicate trajectories. PMCTS samples multiple unique trajectories fully independently in parallel from a theoretically motivated, stochastic selection policy; explicitly controls the entropy of the selection policy, to reduce the number of duplicate trajectories; uses Sequential Monte Carlo (SMC, see [26]) within the framework of MCTS to maintain principled estimates of the expected return with respect to the intended target policy and account for the presence of any remaining duplicate trajectories; uses the Effective Sample Size (ESS, see [26]) to estimate the variance of the SMC expectation estimator and maintain principled updates to the search tree; and last, uses importance sampling to mitigate the effect of over-commitment to catastrophic returns.

We establish *policy improvement* theoretical guarantees for baseline MCTS as well as for PMCTS. Empirically, PMCTS demonstrates strong inference-time scaling, outperforming or comparing well to the popular, heuristic-based baselines across a range of classical RL and MCTS benchmark domains, including hard zero-sum games such as chess and Go, discrete-action, single-agent tasks and classical continuous control.

2 Background

In RL, the environment is represented by a Markov Decision Process [MDP, 27] $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \rho, R, P, \gamma \rangle$. $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions both of which we will consider as finite for presentation and analysis simplicity. ρ is an initial state distribution, $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a bounded possibly stochastic reward function and P is a transition distribution $s' \sim P(\cdot|s, a)$. The policy of the agent $\pi \in \Pi$ is defined as a distribution over actions $a \sim \pi(s)$ with probabilities specified by $\pi(a|s)$. The objective J_π is the maximization of the expected discounted return, or state-value $V^\pi(s)$:

$$J_\pi = \mathbb{E}_\rho\big[V^\pi(s_1)\big] = \mathbb{E}\Big[\textstyle\sum_{t=1}^{H} \gamma^{t-1} R(s_t, a_t)\Big], \quad s_1 \sim \rho,\ a_t \sim \pi(s_t),\ s_{t+1} \sim P(s_t, a_t), \quad (1)$$

over the initial state distribution ρ. The discount factor $0 < \gamma < 1$ is used when $H \to \infty$ to guarantee that V^π remains bounded or for stability and otherwise usually $\gamma = 1$. A state-action ***Q-value function*** is defined as follows: $Q^\pi(s, a) = \mathbb{E}[R(s, a) + \gamma V^\pi(s')]$.

2.1 Search for policy improvement

RL algorithms which have access to a dynamics model of the environment $\mathcal{M} = (P, R)$ (exact or approximate) often use *search* to drive the iterative improvement of the acting policy [28]. Search refers to the process of look-ahead in the model from a state s, often using a DNN-based prior policy π_θ and state value function v_ϕ, in order to extract an improved policy $\pi_{\text{search}}(s)$ at the current state s. A popular example is AlphaZero which uses MCTS for that purpose.

For a policy improvement operator $\mathcal{I} : \Pi \times \mathcal{Q} \to \Pi$, *policy improvement* is formally defined: $\forall s \in \mathcal{S} : V^{\mathcal{I}(\pi,q)}(s) \geq V^\pi(s)$ and there is at least one state where this inequality is strict, unless

π is already optimal. We define $\mathcal{Q}$ generally as the set of all bounded functions on the state-action space $q \in \mathcal{Q} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, to indicate that the operators are defined for exact as well as approximate values. Search algorithms generally return: (I) A selected action a, (II) An improved policy $\pi_{\text{search}}(s)$ and (III) A value $V_{\text{search}}(s) \approx V^{\pi_{\text{search}}}(s)$. $\pi_{\text{search}}(s)$ is used to train π_θ with cross entropy loss and $V_{\text{search}}(s)$ is used to produce bootstraps for TD-targets [4] or even value targets directly [29].

2.2 Monte Carlo Tree Search (MCTS)

The MCTS algorithm maintains a search tree over states with its root $s =: s_1$ representing the current state s in the environment. Traditionally, MCTS is used for search with an exact, deterministic model $\mathcal{M}$ and finite state-action environments. For simplicity of presentation, we will maintain these assumptions hence. However, MCTS has been used successfully for policy improvement and action selection with learned [4] and stochastic transition [30] models as well as in continuous state-action spaces [8, 31]. None of the contributions we make in this paper disagree with these applications. The tree is constructed by iterating (I) *selection*, (II) *expansion* and (III) *backpropagation* M times (often called *number of expansions*, *simulations* or the *search budget*):

Selection: At iteration i, a trajectory $\tau^i_{1,T} := \tau^i_T = (s^i_1, a^i_1, \dots, s^i_T, a^i_T)$ in the tree is selected using a *selection policy* π_i. Traditionally, MCTS uses deterministic selection policies such as π_i^{UCT} or π_i^{PUCT} [1, 2, 23, see Appendix H.1 for additional detail]. π_i uses the current state-action evaluation q_i, the number of prior backpropagation steps $M_i(s_t)$ through each node s_t (and $M_i(s_t, a)$ which can be computed using $M_i(s_{t+1})$ of the node associated with the transition (s_t, a)) and often a *prior policy* π_θ. It has been shown that $\pi_i^{UCT}, \pi_i^{PUCT}$ *track* the policy improvement of specific improvement operators $\mathcal{I}(\pi_\theta, q_i)$ [32]. Further, that using $\mathcal{I}(\pi_\theta, q_i)(s)$ directly rather than $\pi_i^{PUCT}(s)$ results in better performance, for example with $\mathcal{I}$ the *regularized policy improvement* operator $\mathcal{I}_{\text{reg}}$ [24]:

$$\pi_i(a|s_t) = \mathcal{I}_{\text{reg}}(\pi_\theta, q_i)(a|s) \propto \exp(\log \pi_\theta(a|s_t) + \beta_i(s_t) q_i(s_t, a)), \tag{2}$$

$$\beta_i(s_t) = (c_{\text{visit}} + \max_{a \in \mathcal{A}} M_i(s_t, a)) c_{\text{scale}} \tag{3}$$

To compute π_i, evaluation $q_i(s_t, a)$ is needed for *all* actions, including those that have not yet been searched. To achieve this, the *completed Q values* mechanism [24] is used, which replaces $q_i(s_t, a) := v_{\text{mix}}(s) \approx V^{\pi_\theta}(s_t)$, when $M(s_t, a) = 0$ while preserving policy improvement. $v_{\text{mix}}(s) = \sum_{a \in \{a : M(s_t, a) > 0\}} \pi_\theta(a|s_t) q_i(s, a) + \sum_{a \in \{a : M(s_t, a) = 0\}} \pi_\theta(a|s_t) v_\phi(s)$.

Expansion: When selection arrives at an unobserved transition (s_T, a_T) (a *leaf*), it is *expanded*. This gathers information from the model: $r_T = \mathbb{E}[R(s_T, a_T)], s_{T+1} \sim P(s_T, a_T), v_{i+1}(s_{T+1}) \approx V^{\pi_\theta}(s_{T+1})$. In classical MCTS, the new node is evaluated using the average returns of K rollouts in the model, sampled i.i.d. from state s_{T+1} using a *rollout policy*. Following AlphaGo [2], modern variants use a DNN $v_{i+1}(s_{T+1}) := v_\phi(s_{T+1})$ instead.

Backpropagation: The bootstrapped return of the selection trajectory $\nu_i(s_t) = \sum_{j=0}^{T-t} \gamma^j r_j + \gamma^{T-t+1} v_{i+1}(s_{T+1})$ is propagated backwards through all nodes s_t along the trajectory. The nodes maintain the reward to the node $r(s_{t-1}, a_{t-1})$ and an estimate of the average value across $\pi_1, \dots, \pi_i$:

$$v_{i+1}(s_t) = \tfrac{1}{M_i(s_t)} \sum_{j=1}^{M_i(s_t)} \nu_j(s_t), \quad q_{i+1}(s_t, a_t) = r(s_t, a_t) + \gamma v_{i+1}(s_{t+1}). \tag{4}$$

MCTS returns $\pi_{\text{search}}(s) = \mathcal{I}_{\text{reg}}(\pi_\theta, q_M)(s)$ and $V_{\text{search}}(s) = \sum_{a \in A} \pi_{\text{search}}(a|s) q_M(s, a)$. To act in the environment, an action can be sampled from $\pi_{\text{search}}(s)$ (see appendix H.2 for more detail).

2.3 Sequential Monte Carlo for search for policy improvement in RL

SMC methods use *sequential importance sampling* (SIS) to approximate a sequence of *target distributions* using *proposal distributions* [26]. In RL, SMC methods have been adapted for search for policy improvement [33] by formulating the proposal and target as distributions over trajectories τ_t using π_θ as the *proposal policy* and $\pi'(s_t) = \mathcal{I}(\pi_\theta, Q^{\pi_\theta})(s_t)$ as the *target policy* [34]. At each time step $t \in \{1, \dots, T\}$, N particles comprised of a trajectory and weights $\{\tau^n_t, w^n_t\}_{n=1}^N$ are updated via *sampling* and *correction*. *Sampling*: each trajectory τ^n_{t-1} is extended by sampling $s^n_{t+1} \sim P(\cdot|s^n_t, a^n_t), a^n_t \sim \pi_\theta(\cdot|s^n_t), s^n_1 = s$, where s is the current state in the environment. *Correction*: The weights are corrected using SIS (left) such that the weighted particles $\{\tau^n_t, w^n_t\}_{n=1}^N$ provide

consistent approximation of expectations under the target:

$$w_t^n = w_{t-1}^n \frac{\pi'(a_t^n|s_t^n)}{\pi_\theta(a_t^n|s_t^n)}, \quad \frac{1}{\sum_{n=1}^N w_t^n} \sum_{n=1}^N w_t^n \left(\mathcal{R}(\tau_t^n) + \gamma^t V^{\pi_\theta}(s_{t+1}^n)\right) \xrightarrow[N\to\infty]{\mathbb{P}} V^{\pi'}(s_1). \quad (5)$$

We refer to [34] for derivation and Appendix C.3 and [26] for more detail.

3 Particle Monte Carlo Tree Search

We design Particle Monte Carlo Tree Search (PMCTS) to achieve three desiderata: (I) Theoretically motivated selection and evaluation, to enable principled policy improvement. (II) Selection of multiple unique trajectories fully independently in parallel, to support batch-parallelization. (III) Correctly account for the presence of remaining duplicate trajectories, to provide principled updates to the tree. We begin by designing a simple, principled, particle-based batch-parallel variant of MCTS, which preserves policy improvement under *classical* assumptions (namely, unbiased and uncorrelated evaluations). Next, we identify the challenges that arise from using DNN evaluations which do not satisfy these classical assumptions. We then proceed to present PMCTS, which addresses these challenges and achieves the set desiderata.

3.1 A simple, parallel, particle based Monte Carlo Tree Search algorithm

We begin with an observation: the principled stochastic selection policies proposed by [32], such as π_i (Equation 2), enable the principled selection of multiple unique trajectories fully independently in parallel through i.i.d. sampling from π_i. Motivated by this observation, we design a simple particle-based MCTS algorithm which preserves the policy improvement properties of MCTS and reduces evaluation variance. The algorithm operates the same as baseline MCTS, except that each step is now batch-parallelized over particles.

Particle-based selection: At each iteration i, N trajectories are sampled independently in parallel from the stochastic selection policy π_i (Equation 2).

Particle-based expansion: N nodes s_{T+1}^n are expanded in batch-parallel:

$$\left\{s_{T+1}^n \sim P(\cdot|s_T^n, a_T^n), \quad r(s_T^n, a_T^n) = \mathbb{E}[R(s_T^n, a_T^n)], \quad v_i(s_{T+1}^n), \quad \pi_\theta(s_{T+1}^n)\right\}_{n=1}^N. \quad (6)$$

Particle-based backpropagation: We now need to compute the particle-based return ν_i and backpropagation v_i. Under classical assumptions, the leaf evaluations $v_i(s_{T+1}^n)$ are unbiased and independently sampled and the average of the returns of the particles that passed through s_t (Equation 7 left) provides an unbiased estimate of $V^{\pi_i}(s_t)$. To compute the backup $v_i(s_t)$ we use $1/N(s_t)$ in the optimal estimator of the mean (the *inverse-variance weighting* [35], Equation 7 right, see Appendix C.1 for derivation). $N(s_t)$ is the number of particles passing through the node this iteration and is inversely proportional to the variance $\mathbb{V}[\nu_i(s_t)]$:

$$\nu_i(s_t) = \frac{1}{N(s_t)} \sum_{n=1}^N \mathbb{1}_{s_t \in \tau_T^n} \nu^n(s_t), \quad v_{i+1}(s_t) = v_i(s_t) + \frac{(\nu_i(s_t) - v_i(s_t))N(s_t)}{M_{i+1}(s_t)}, \quad (7)$$

and $M_{i+1}(s_t) = M_i(s_t) + N(s_t)$. We summarize Simple PMCTS in Algorithm 1 in Appendix B. Action selection and policy improvement follow the description in Section 2, see Appendix H.2 for more detail. Simple PMCTS is principled and batch-parallelizable, addressing the first and second limitations of prior methods, but does not yet account for the combined effect of duplicate trajectories and DNN evaluations.

3.2 Enter Deep Neural Networks

With DNN evaluations, the returns $\nu_i^n(s_t)$ cannot be assumed to be entirely uncorrelated or unbiased. This is a general problem in search algorithms such as MCTS, SMC for search [33, 34] and MPPI [36]. Despite this, these algorithms are extremely successful in practice, implicitly assuming that *unique* evaluations are *sufficiently* uncorrelated and unbiased. In discrete-action, parallel search algorithms this problem is significantly exacerbated. *Multiple* parallel selection trajectories can arrive at *the same*

leaf s_{T+1}, even when π_i is stochastic. In this case, all trajectories where $s^n_{T+1} = s_{T+1}$ will have *fully* correlated returns. This causes two main problems: (I) Duplicate leaf evaluations do not reduce the evaluation's variance and waste compute. (II) The variance $\mathbb{V}[\nu_i(s_t)]$ cannot be assumed to be proportional to $1/N(s_t)$ resulting in *incorrectly weighted backpropagation*. We proceed to extend Simple PMCTS with mechanisms that address problems (I-II) and design the complete PMCTS algorithm.

3.3 Weighted Particle Monte Carlo Tree Search (PMCTS)

Addressing problems (I-II) requires the algorithm to reduce the number of duplicate trajectories while accounting for the remaining duplicate trajectories and maintaining a principled estimate of V^{π_i}. To achieve that, we use SMC within PMCTS, with four mechanisms: *higher-entropy sampling* to increase the number of unique trajectories; *Deduplication* to mitigate the effect of remaining duplicate trajectories and maintain a principled estimate of V^{π_i}; And the *Effective Sample Size (ESS)* [26] for principled backpropagation. In addition, SMC allows the algorithm to address a more general problem to MCTS, over-commitment to catastrophic returns, through a fourth mechanism we call *retrospective reweighting*. The resulting algorithm is PMCTS, which is described below. PMCTS maintains the same structure of particle-based selection, expansion, backpropagation, except that each particle n is now associated with a weight w^n_t which is updated during selection.

Weighted particle-based selection with *higher-entropy sampling*: To increase trajectory diversity actions are sampled from a higher-entropy proposal policy $\hat{\pi}_i$ rather than π_i. The particle's weight is computed using SIS to maintain a principled estimate of the expectation with respect to π_i:

$$w^n_t = w^n_{t-1} \frac{\pi_i(a^n_t \mid s^n_t)}{\hat{\pi}_i(a^n_t \mid s^n_t)}, \qquad \hat{\pi}_i(a_t|s_t) \propto \pi_i(a_t|s_t)^{\frac{1}{\eta(s_t)}}. \tag{8}$$

This mechanism allows the algorithm to minimize the variance of the update by trading off between two sources of variance: the effect of $\hat{\pi}_i$ diverging from π_i and the effect of reduced duplicates.

For simplicity, in this work we use a constant rather than a dynamic $\eta(s) := \eta$. In general, any policy $\hat{\pi}_i$ that is absolutely continuous with respect to π_i can be used. If particles are allowed to communicate at the cost of parallelization the temperature can be chosen based on the number of particles per node to reduce variance. SMC resampling techniques can additionally be incorporated for further variance reduction [26]. However, resampling reduces trajectory diversity, requires particles to communicate and can insert bias, and we leave its investigation for future work.

Weighted particle-based backpropagation with *deduplication* and *ESS*: After expansion, PMCTS aggregates all duplicate trajectories into a single particle while preserving the correct total weight: $w^m_T(s_{T+1}) = \sum_{n=1}^N w^n_T \mathbb{1}_{s_{T+1}=s^n_{T+1}}$, which we call *deduplication*. This results in a new set of particles where all trajectories are unique while correctly preserving the estimate of the expectation with respect to π_i. We will refer to this new particle set using the same notation $\{\tau^n_T, w^n_T\}_{n=1}^N$. The backup $\nu_i(s_t)$ is computed using the particles' normalized weights $\bar{w}^n_T(s_t)$ and returns $\nu^n(s_t)$:

$$\nu_i(s_t) = \sum_{n=1}^N \underbrace{\bar{w}^n_T(s_t)\nu^n(s_t)\mathbb{1}_{s_t \in \tau^n_T}}_{\text{Weighted particle returns}}, \qquad \bar{w}^n_T(s_t) = \frac{w^n_T \mathbb{1}_{s_t \in \tau^n_T}}{\underbrace{\sum_{m=1}^N w^m_T \mathbb{1}_{s_t \in \tau^m_T}}_{\text{Normalized per node}}}. \tag{9}$$

To accurately estimate the update to the average v_{i+1} we need to estimate the variance $\mathbb{V}[\nu_i(s_t)]$. A standard choice is the inverse ESS [26] $\mathbb{V}\left[\nu_i(s_t)\right] \approx \frac{1}{\text{ESS}(s_t)} = \sum_{n=1}^N (\bar{w}^n_T(s_t))^2$. Thus:

$$v_{i+1}(s_t) = v_i(s_t) + \frac{(\nu_i(s_t) - v_i(s_t))\text{ESS}(s_t)}{M_{i+1}(s_t)}, \qquad M_{i+1}(s_t) = M_i(s_t) + \text{ESS}(s_t), \tag{10}$$

and $M(s_{T+1}) = 1$. See Appendix C.2 for more detail and derivation.

Retrospective reweighting: SMC additionally enables PMCTS to address a third problem, general to MCTS: over-commitment to catastrophic returns [see 37]. At parent s_T MCTS samples $a_T \sim \pi_i(s_T)$ leading to evaluation $q_i(s_T, a_T) = r_T + \gamma v_\phi(s_{T+1})$. The action a_T may be catastrophic in terms of its true value $Q^{\pi_\theta}(s_T, a_T)$ (for example, falling off a cliff), which cannot be known in advance: the purpose of the expansion is to gather new information. In standard MCTS variants, this catastrophic

evaluation will then be backpropagated to the root, unnecessarily contaminating all nodes along the trajectory even if the catastrophic action is entirely avoidable. To address this problem, after the expansion step and before backpropagation we first compute the "retrospective"-target policy $\pi_{i+1}(s_T)$ by computing $q_{i+1}(s_T)$ and then use π_{i+1} instead of π_i to compute w_T^n:

$$w_T^n = w_{T-1}^n \frac{\pi_{i+1}(a_T^n \mid s_T^n)}{\hat{\pi}_i(a_T^n \mid s_T^n)}, \quad \pi_{i+1}(a_T|s_T) \propto \pi_\theta(a_T|s_T) \exp\big(\underbrace{\beta_{i+1}(s_T) q_{i+1}(s_T, a_T)}_{\text{Post-expansion.}}\big). \quad (11)$$

The weight w_T^n now accounts for the new evaluation q_{i+1}. As a result, at any node s_t with more than one particle $N_i(s_t) > 1$ the aggregation of catastrophic trajectories is reduced.

This completes the PMCTS algorithm, which remains principled and batch parallelizable while accounting for the effects of duplicate trajectories. We summarize PMCTS in natural language in Appendix B.2 for added clarity, along with pseudocode in Algorithm 2.

4 Theoretical Analysis

The main use case of MCTS is for policy improvement: either explicitly to produce an improved policy in an RL training loop, or implicitly to select better actions by sampling from an improved policy. For this reason, our analysis focuses on establishing that PMCTS is a principled policy improvement operator.

[32] have shown that MCTS with π^{PUCT} *tracks* policy improvement and proposed to use a principled improved policy such as π_i (Equation 2), the policy tracked by π^{PUCT}, as the search policy, showing that MCTS can be *interpreted* as a policy improvement operator. We extend this result, showing that the MCTS variant which uses π_i *guarantees* policy improvement at every iteration under classical assumptions and with respect to the expected-backpropagation. That is, when $\nu_i(s_t) \leftarrow \mathbb{E}\big[\mathcal{R}(\tau_{t,T}) + V^{\pi_\theta}(s_{T+1})\big]$. The classical assumptions under which MCTS is traditionally analyzed are that of i.i.d., unbiased and finite-variance node evaluations at the leaves and a deterministic and accurate dynamics model.

Theorem 1 (MCTS as a Principled Policy Improvement Operator)**.** *(Informal) Under the classical assumptions, with π_i (Equation 2) as the selection policy and with respect to the expected backpropagation, MCTS guarantees policy improvement at every iteration.*

Intuition: In expectation on each backpropagation step, ν_i is the true value of the selection policy, which is an improved policy with respect to π_θ. The value at each node q_i is the average across iterations and is thus also the value of a *mixture* across the historical policies $\pi_1(s), \ldots, \pi_i(s)$. A policy which is a mixture of improved policies is an improved policy, and policy improvement with respect to the value of an *improved* policy maintains policy improvement [34]. We include the formal theorem and proof by induction over iterations in Appendix D.1.

Analyzing MCTS with respect to the *expected backpropagation* is necessary because in every iteration MCTS conducts policy improvement (i.e. maximizes the policy) with respect to *observed* returns rather than the true value and thus suffers from over-estimation bias. As a result, policy improvement is only guaranteed with respect to the expectation on the backpropagation step itself, rather than on the entire search. The notion of the expected backpropagation is somewhat artificial in standard MCTS: it corresponds to an expansion step which expands *all* leaves in the tree at *every* iteration with accurate evaluations, and correctly weighting them under π_i, which is not how the algorithm works in practice. In Simple PMCTS and PMCTS however, this notion is natural, as it directly corresponds to the limit of $N \to \infty$. In fact, baseline MCTS can be interpreted as single-particle Simple PMCTS. This allows us to establish a stronger result, that Simple PMCTS and PMCTS are both principled policy improvement operators under the more natural notion of $N \to \infty$.

Theorem 2 (Simple PMCTS is a Principled Policy Improvement Operator)**.** *(Informal) Under the classical assumptions and in the limit of $N \to \infty$, Simple PMCTS guarantees policy improvement at every iteration almost-surely.*

Intuition: The backpropagation estimates of Simple PMCTS converge *almost-surely* (i.e. $\mathrm{P}\left(\lim_{N\to\infty} V_N(s) = V^{\pi_i}(s)\right) = 1$) to the expected backpropagation values as $N \to \infty$, and thus the same proof as applies almost-surely. See Appendix D.2 for the formal Theorem and its proof.

Theorem 3 (PMCTS is a Principled Policy Improvement Operator)**.** *(Informal) Under the classical assumptions and in the limit of $N \to \infty$, PMCTS guarantees policy improvement at every iteration almost-surely.*

Intuition: The SMC estimates used by PMCTS converge almost-surely to the expected backpropagation values as $N \to \infty$. Deduplication preserves the importance-weighted particle distribution, while the ESS-based weighting becomes exact in the limit. Thus, almost surely, the same proof as Theorem 1 applies to PMCTS. See Appendix D.3 for the formal Theorem and its proof.

We additionally include a runtime complexity analysis in Appendix E.

5 Related Work

Runtime-inference scaling of MCTS has been a long time objective of AI research [17]. The canonical approaches are (I) *root parallelization* (running different instances of MCTS in parallel) and (II) *leaf parallelization* (scaling the evaluation of leaves with parallel compute). Both rely on stochasticity in the selection and / or evaluation to scale and are thus unsuited for runtime inference scaling with modern DNN evaluations and deterministic dynamics and selection. (III) *Tree parallelization* (parallelizing the search itself), which canonically uses the *virtual visits* heuristic to incentivize the selection of multiple unique trajectories and circumvent these challenges. When a process arrives at action a in node s it increments a *virtual visit* $M_i^{\text{virtual}}(s, a)$ [18]. When another process arrives at s it computes the deterministic selection policy $\pi_i^{\text{PUCT}}(s)$ using $M_i^{\text{virtual}}(s, a)$ and a heuristic which effectively *predicts* the evaluation of the previous processes in the hope that π_i^{PUCT} selects a different action. The selected trajectories are then evaluated in parallel, usually batched, and backpropagated, whether they are unique or not, resulting in incorrectly weighted estimates of the evaluation's variance (when DNNs are used for evaluation) as well as an unclear interpretation of the policy being evaluated. In addition, to work as intended this selection requires the processes to operate *sequentially* at each *shared* node, both preventing effective batched-parallelization as well as in the worst case resulting in entirely sequential–rather than parallel–selection (See Appendices F and H.3 for more detail). In contrast, PMCTS is both principled (Theorem 3) and selects trajectories fully independently in parallel, enabling fully-batched parallelization of the selection step.

Adjacent to MCTS and PMCTS, SMC has been used directly for search for policy improvement [33, 34, 38]. The resulting SMC-based search algorithms are fundamentally different to MCTS: They maintain statistics only over N particles rather than memorize the entire search tree. As a result, they can only converge to the value of an *improved policy* in the limit of infinite search horizon [34], rather than the optimal policy in the case of MCTS [1]; they also struggles to scale with large search horizons without large particle budgets [34] as we also demonstrate in Appendix G; And they are not designed to account for duplicate trajectories. In contrast, PMCTS memorizes the entire tree and performs well with large search budgets irrespective of the number of particles. PMCTS is additionally designed to account for the presence of duplicate trajectories with the three mechanisms *increased-entropy selection*, *deduplication* and *the ESS* which are novel to this line of work.

6 Experiments

Our objective in this work is to present a novel and effective tree-parallel MCTS algorithm which is both principled as well as suited for batch-parallelization and runtime scaling on the GPU. In this section, we evaluate PMCTS empirically across three popular MCTS and RL benchmark domains and one LLM inference scaling experiment. These encompass: (I) Zero-sum board games (*chess, 19x19 Go, 9x9 Go* and *Gardner chess)* [39]; (II) Discrete-action popular [34, 38] single actor environments (*Snake* and *RubiksCube* [40]); (III) Classical continuous control environments (*Ant, HalfCheetah* and *Humanoid* [41]), where MCTS is extended to continuous actions in the manner of [31] (see Appendix I.5 for more detail); And (IV) scaling the inference of an LLM solving different *Sokoban* puzzles. As performance metrics, we use the popular Bayes Elo [42] metric (measures strength of play in zero-sum games), the observed averaged return and number of Sokoban instances solved, each where relevant. To aggregate across different environments, we take the averaged Bayes-elo, average normalized return and averaged number of instances solved, respectively.

To evaluate PMCTS as a tree-parallel MCTS algorithm, we begin by comparing the inference-time scaling of PMCTS with number of particles N to the two standard and popular tree-parallel MCTS variants, Virtual Means and Virtual Losses. In each environment, all agents use the same π_θ and v_ϕ DNNs in the search and the default, popular MCTS hyperparameters [24, 43]. The only new parameter introduced with this work is the temperature η, whose effect is investigated in Figure 3, based on which we choose $\eta = 1.5$ which is used in all experiments. For additional experimental and implementation details see Appendices H and I.

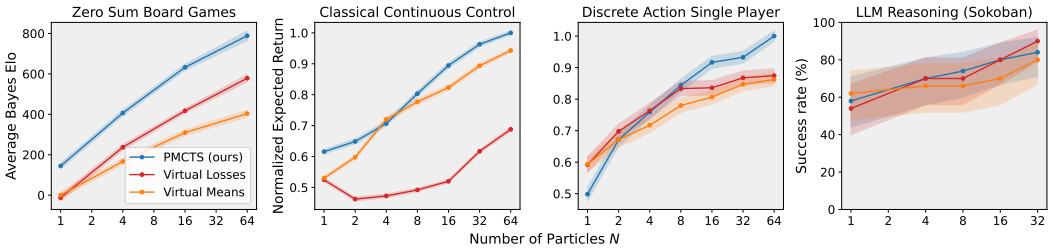


Figure 1: Scaling of tree-parallel MCTS variants with parallel compute (number of particles N) across different domains. Bayes Elo with 95% CI, in board games and 95% Gaussian CI in discrete and continuous and 95% Wilson CI in Sokoban.

PMCTS consistently scales with N in all domains; Outperforms or performs comparably (no statistically significant difference) to both baselines, at every parallelization budget $N > 1$, in every domain; And is the only method that is consistently dominant across all four domains. We additionally investigate the scaling of the different tree-parallel algorithms in an example environment across different parallel-simulation budgets M, in Figure 4 in Appendix G. PMCTS scales well across increasing M and outperforms both baselines at every budget pair M and N.

To validate that PMCTS indeed provides effective runtime scaling with batch parallelization we investigate the wallclock runtime scaling of the different tree-parallel MCTS variants with vectorized, synchronously parallel implementations suited for batch-parallelization, in Figure 2 left and center. PMCTS's runtime scales very well with additional particles, as expected from an algorithm suited for batch parallelization. In contrast, the synchronous implementations of Virtual Means and Virtual Losses do not scale well, as expected, due to their inability to select multiple unique trajectories fully independently in parallel and subsequent collapse to sequential rather than parallel selection (see Appendix F.1 for additional detail).

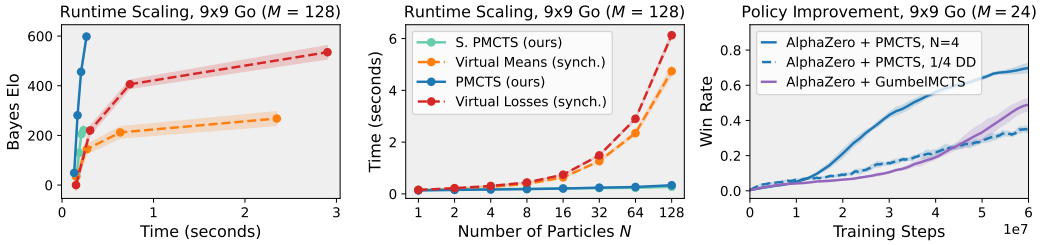


Figure 2: **Left:** Runtime scaling, Bayes Elo with 95% CI, $N = (1, 4, 16, 64)$ plotted. **Center:** Runtime scaling, 95% confidence interval across repeated evaluations. **Right:** Win rate vs. frames during training of AlphaZero with PMCTS and Gumbel MCTS, mean and 95% CI across 3 seeds.

To investigate the benefits of PMCTS to an RL training loop, we investigate scaling policy improvement during training with PMCTS, in Figure 2 right. We compare to GumbelMCTS [24], a modern non-parallel MCTS baseline. PMCTS with $N = 4$ outperforms baseline Gumbel MCTS in sample efficiency, demonstrating better policy improvement. During training, one may trade off parallel compute for *increased data diversity* (root-parallelization across different states) or for *better policy targets* (tree-parallelization in the same state). We include an example of this tradeoff with the dashed-line agent, which trades off increased $N = 4$ and reduced data generated during training by $1/4$. All other training parameters, which were tuned for the baseline agent, are kept the same. This agent performs similarly in sample efficiency to the baseline in this example.

To validate that each mechanism developed in Section 3.2 indeed contributes to the overall performance of the algorithm, we ablate the different mechanisms in sequence on 9x9 Go ($M = 128$),

in Figure 3 left. A very large number of mechanism combinations is possible. To maintain an informative and reasonable number of experiments, we compare from Simple PMCTS all the way to full PMCTS, with the different mechanisms activated in order, resulting in 6 different agents, each having one more mechanism activated than the previous. These are: (I) **S**imple **PMCTS** (Algorithm 1, shortened to **S**). (II) + **D**eduplication which removes duplicate trajectories. Deduplication computes the weight of the update (Equation 7, right) using the number of *unique* trajectories instead of $N(s_t)$. (III) + **E**SS which replaces the weight of the update with a direct estimate of the evaluation's variance, 1 / ESS (Equation 10). (IV) + **T**emperature, which increases the number of unique trajectories by increasing the entropy of the selection policy using the temperature τ (Equation 8, right). (V) + weight **C**orrection which uses SIS (Equation 8, left) to estimate the expected return with respect to π_i rather than $\hat{\pi}_i$. (VI) The **PMCTS** algorithm, which uses retrospective reweighting in addition to all prior mechanisms. As expected, performance increases consistently with every additional feature.

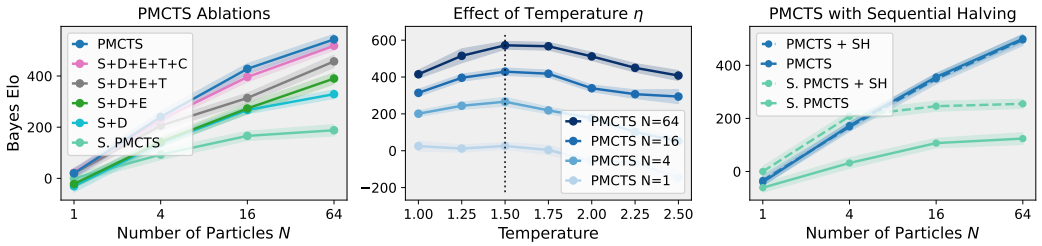


Figure 3: Ablations and hyperparameter evaluation on 9x9 Go ($M = 128$). Bayes Elo with 95% CI.

To investigate the effect of the introduced temperature hyperparameter η, we investigate the effect of different η on the performance of PMCTS across different N in Figure 3 center. The dominating temperature is 1.5, the temperature used in all other experiments. Sequential Halving (SH, [44]) has been used by [24, 34] to improve search at the root. We investigate its effect on PMCTS, in Figure 3 right. SH, which enforces particle diversity at the root, has a strong positive effect on *Simple* PMCTS. However, when combined with PMCTS, is has no additional effects on the scaling. To reduce unnecessary complexity, we use PMCTS without SH in all experiments.

We include several additional results in Appendix G. These include the individual results aggregated in Figure 1; An example comparison to *root-parallelization*, to demonstrate that indeed root-parallelization is not effective for runtime scaling with DNN evaluations, as discussed in Sections 1 and 5; A comparison to the non-MCTS, SMC-based search variants, demonstrating their limitations compared to parallel-MCTS variants; And an investigation of the manner in which the temperature η allows PMCTS to minimize evaluation variance by trading off between increased number of unique trajectories and reduced divergence between the proposal $\hat{\pi}_i$ and target π_i.

7 Conclusions and Future Work

In this work, we introduced *Particle MCTS* (PMCTS), a principled, tree-parallel MCTS algorithm suited for runtime scaling through parallel compute, batch-parallelization and GPU acceleration. PMCTS uses stochastic selection policies and Sequential Monte Carlo (SMC) to select and evaluate multiple unique trajectories independently in parallel while maintaining principled evaluation with respect to the target policy. We established that MCTS and PMCTS are both principled policy improvement operators. Empirically, PMCTS scales well with parallel compute and significantly outperforms prior, heuristic based approaches across a wide range of benchmark domains, including two-player zero-sum board games, single player discrete action and classical continuous control domains. PMCTS opens up multiple exciting venues for future work such as more advanced temperature schedules for better proposal policies, incorporating proposals that sample without replacement to increase the number of unique trajectories and resampling methods [26] for different variance tradeoffs.

Acknowledgments and Disclosure of Funding

This work used the Dutch national e-infrastructure with the support of the SURF Cooperative using grant no. EINF-13454. Research reported in this work was partially or completely facilitated by com-

putational resources and support of the Delft AI Cluster (DAIC) at TU Delft (RRID: SCR_025091), but remains the sole responsibility of the authors, not the DAIC team.

References

[1] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *The 17th European Conference on Machine Learning*, 2006. doi: 10.1007/11871842_29.

[2] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961.

[3] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018. doi: 10.1126/science.aar6404.

[4] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020. doi: 10.1038/s41586-020-03051-4.

[5] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J. R. Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and Pushmeet Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022. doi: 10.1038/s41586-022-05172-4.

[6] Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Koppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023. doi: 10.1038/s41586-023-06004-9.

[7] Amol Mandhane, Anton Zhernov, Maribeth Rauh, Chenjie Gu, Miaosen Wang, Flora Xue, Wendy Shang, Derek Pang, Rene Claus, Ching-Han Chiang, et al. Muzero with self-competition for rate control in vp9 video compression. *arXiv preprint arXiv:2202.06626*, 2022.

[8] Shengjie Wang, Shaohuai Liu, Weirui Ye, Jiacheng You, and Yang Gao. Efficientzero V2: mastering discrete and continuous control with limited data. In *Forty-first International Conference on Machine Learning*, 2024.

[9] Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. doi: 10.18653/V1/2023.EMNLP-MAIN.507.

[10] Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *Forty-first International Conference on Machine Learning*, 2024.

[11] Ziyu Wan, Xidong Feng, Muning Wen, Stephen Marcus McAleer, Ying Wen, Weinan Zhang, and Jun Wang. Alphazero-like tree-search can guide large language model decoding and training. In Ruslan Salakhutdinov, Zico Kolter, Katherine A. Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Forty-first International Conference on Machine Learning*, 2024.

[12] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Yang Wang. SWE-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations*, 2025.

[13] Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. Rest-mcts*: Llm self-training via process reward guided tree search. *The 37th Annual Conference on Advances in Neural Information Processing Systems*, pages 64735–64772, 2024.

[14] Zijing Shi, Meng Fang, and Ling Chen. Monte carlo planning with large language model for text-based game agents. In *The Thirteenth International Conference on Learning Representations*, 2025.

[15] Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. In *Forty-second International Conference on Machine Learning*, 2025.

[16] Kou Misaki, Yuichi Inoue, Yuki Imajuku, So Kuroki, Taishi Nakamura, and Takuya Akiba. Wider or deeper? scaling LLM inference-time compute with adaptive branching tree search. *arXiv preprint arXiv:2503.04412*, 2025.

[17] Tristan Cazenave and Nicolas Jouandeau. On the parallelization of UCT. In *Computer Games Workshop 207 (CGW07)*, 2007.

[18] Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In *6th International Conference on Computers and Games (CG 2008)*, 2008.

[19] Xiufeng Yang, Tanuj Kr Aasawat, and Kazuki Yoshizoe. Practical massively parallel monte-carlo tree search applied to molecular design. In *The 9th International Conference on Learning Representations*, 2021.

[20] Lars Schäfers. *Parallel Monte-Carlo tree search for HPC systems and its application to computer go*. PhD thesis, University of Paderborn, 2014.

[21] Anji Liu, Yitao Liang, Ji Liu, Guy Van den Broeck, and Jianshu Chen. On effective parallelization of monte carlo tree search. *CoRR*, abs/2006.08785, 2020.

[22] Tristan Cazenave. Batch monte carlo tree search. In *Computers and Games - International Conference*, 2022. doi: 10.1007/978-3-031-34017-8_13.

[23] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[24] Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with Gumbel. In *The Tenth International Conference on Learning Representations*, 2022.

[25] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/jax-ml/jax`.

[26] Nicolas Chopin and Omiros Papaspiliopoulos. *An Introduction to Sequential Monte Carlo*. Springer Series in Statistics. Springer, Cham, 1st edition, 2020. doi: 10.1007/978-3-030-47845-2.

[27] Richard Bellman. A Markovian Decision Process. *Journal of Mathematics and Mechanics*, 6 (5):679–684, 1957.

[28] Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Model-based Reinforcement Learning: A Survey. *Foundations and Trends® in Machine Learning*, 16(1): 1–118, 2023. doi: 10.1561/2200000086.

[29] Yaniv Oren, Viliam Vadocz, Matthijs T. J. Spaan, and Wendelin Boehmer. Epistemic Monte Carlo Tree Search. In *The Thirteenth International Conference on Learning Representations*, 2025.

[30] Ioannis Antonoglou, Julian Schrittwieser, Sherjil Ozair, Thomas K. Hubert, and David Silver. Planning in stochastic environments with a learned model. In *The Tenth International Conference on Learning Representations*, 2022.

[31] Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Mohammadamin Barekatain, Simon Schmitt, and David Silver. Learning and Planning in Complex Action Spaces. In *The 38th International Conference on Machine Learning*, 2021.

[32] Jean-Bastien Grill, Florent Altché, Yunhao Tang, Thomas Hubert, Michal Valko, Ioannis Antonoglou, and Remi Munos. Monte-Carlo Tree Search as Regularized Policy Optimization. In *The 37th International Conference on Machine Learning*, 2020.

[33] Alexandre Piché, Valentin Thomas, Cyril Ibrahim, Yoshua Bengio, and Chris Pal. Probabilistic planning with sequential monte carlo methods. In *The 7th International Conference on Learning Representations*, 2019.

[34] Yaniv Oren, Joery A de Vries, Pascal R van der Vaart, Matthijs TJ Spaan, and Wendelin Böhmer. Twice sequential monte carlo for tree search. *The 43rd International Conference on Machine Learning*, 2026.

[35] Joachim Hartung, Guido Knapp, and Bimal K Sinha. *Statistical meta-analysis with applications*. John Wiley & Sons, 2008.

[36] Nicklas Hansen, Hao Su, and Xiaolong Wang. TD-MPC2: scalable, robust world models for continuous control. In *The Twelfth International Conference on Learning Representations*, 2024.

[37] Albin Jaldevik. General tree evaluation for AlphaZero. Master's thesis, Delft University of Technology, 2024.

[38] Joery A. de Vries, Jinke He, Yaniv Oren, and Matthijs T. J. Spaan. Trust-Region Twisted Policy Improvement. In *The 42 International Conference on Machine Learning*, 2025.

[39] Sotetsu Koyamada, Shinri Okano, Soichiro Nishimori, Yu Murata, Keigo Habara, Haruka Kita, and Shin Ishii. Pgx: Hardware-accelerated parallel game simulators for reinforcement learning. In *The 36th Annual Conference on Advances in Neural Information Processing Systems*, 2023.

[40] Clément Bonnet, Daniel Luo, Donal John Byrne, Shikha Surana, Sasha Abramowitz, Paul Duckworth, Vincent Coyette, Laurence Illing Midgley, Elshadai Tegegn, Tristan Kalloniatis, Omayma Mahjoub, Matthew Macfarlane, Andries Petrus Smit, Nathan Grinsztajn, Raphael Boige, Cemlyn Neil Waters, Mohamed Ali Ali Mimouni, Ulrich Armel Mbou Sob, Ruan John de Kock, Siddarth Singh, Daniel Furelos-Blanco, Victor Le, Arnu Pretorius, and Alexandre Laterre. Jumanji: a diverse suite of scalable reinforcement learning environments in JAX. In *The Twelfth International Conference on Learning Representations*, 2024.

[41] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[42] Rémi Coulom. Bayesian Elo Rating. `https://www.remi-coulom.fr/Bayesian-Elo/`, 2005. [Online; accessed 02-05-2024].

[43] DeepMind, Igor Babuschkin, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, Trevor Cai, Aidan Clark, Ivo Danihelka, Antoine Dedieu, Claudio Fantacci, Jonathan Godwin, Chris Jones, Ross Hemsley, Tom Hennigan, Matteo Hessel, Shaobo Hou, Steven Kapturowski, Thomas Keck, Iurii Kemaev, Michael King, Markus Kunesch, Lena Martens, Hamza Merzic, Vladimir Mikulik, Tamara Norman, George Papamakarios, John Quan, Roman Ring, Francisco Ruiz, Alvaro Sanchez, Laurent Sartran, Rosalia Schneider, Eren Sezener, Stephen Spencer, Srivatsan Srinivasan, Miloš Stanojević, Wojciech Stokowiec, Luyu Wang, Guangyao Zhou, and Fabio Viola. The DeepMind JAX Ecosystem, 2020. URL `https://github.com/google-deepmind`.

[44] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost Optimal Exploration in Multi-Armed Bandits. In *The 30th International Conference on Machine Learning*, 2013.

[45] Gabriel Cardoso, Sergey Samsonov, Achille Thin, Eric Moulines, and Jimmy Olsson. BR-SNIS: bias reduced self-normalized importance sampling. *The 35th Annual Conference on Advances in Neural Information Processing Systems*, 2022.

[46] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786–793, 1973. doi: 10.1109/TC.1973.5009159.

[47] G.E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11): 1526–1538, 1989. doi: 10.1109/12.42122.

[48] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. A difference in efficiency between synchronous and asynchronous systems. In *The 13th Annual ACM Symposium on Theory of Computing*, 1981. doi: 10.1145/800076.802466.

[49] S. Ali Mirsoleimani, Aske Plaat, H. Jaap van den Herik, and Jos Vermaseren. An analysis of virtual loss in parallel MCTS. In *The 9th International Conference on Agents and Artificial Intelligence*, 2017. doi: 10.5220/0006205806480652.

[50] Markus Enzenberger and Martin Müller. A lock-free multithreaded monte-carlo tree search algorithm. In *The 12th International Conference on Advances in Computer Games*, 2009. doi: 10.1007/978-3-642-12993-3_2.

[51] S. Ali Mirsoleimani, H. Jaap van den Herik, Aske Plaat, and Jos Vermaseren. A lock-free algorithm for parallel MCTS. In *The 10th International Conference on Agents and Artificial Intelligence*, 2018.

[52] Emil Malmsten and Wendelin Böhmer. Transzero: Parallel tree expansion in muzero using transformer networks. *arXiv preprint arXiv:2509.11233*, 2025.

[53] Scott Cheng, Mahmut T. Kandemir, and Ding-Yong Hong. Speculative monte-carlo tree search. In *The 38th Annual Conference on Neural Information Processing Systems*, 2024.

[54] Juhwan Kim, Byeongmin Kang, and Hyungmin Cho. Specmcts: Accelerating monte carlo tree search using speculative tree traversal. *IEEE Access*, 9:142195–142205, 2021. doi: 10.1109/ACCESS.2021.3120384.

[55] Li-Cheng Lan, Wei Li, Ting-Han Wei, and I-Chen Wu. Multiple policy value monte carlo tree search. In *The Twenty-Eighth International Joint Conference on Artificial Intelligence*, 2019. doi: 10.24963/IJCAI.2019/653.

[56] Yaniv Oren, Moritz A Zanger, Pascal R Van der Vaart, Mustafa Mert Çelikok, Matthijs TJ Spaan, and Wendelin Boehmer. Value Improved Actor Critic Algorithms. In *The 39th Annual Conference on Neural Information Processing Systems*, 2025.

[57] David R Hunter. Mm algorithms for generalized bradley-terry models. *The annals of statistics*, 32(1):384–406, 2004.

Appendix Contents

A Acronym and Symbols List **16**

B Pseudocode **16**

B.1 Simple PMCTS 16

B.2 PMCTS 17

C Derivations **18**

C.1 Derivation of the numerically stable weighted average 18

C.2 Derivation of the particle-based backpropagation step in PMCTS 18

C.3 Approximating expectations under the target in Sequential Monte Carlo 19

C.4 Unbiased estimates of the value at any depth 19

D Formal Theoretical Analysis **20**

D.1 MCTS as a principled policy improvement operator 20

D.2 Simple PMCTS is a principled policy improvement operator 22

D.3 PMCTS is a principled policy improvement operator 22

E Worst-Case Complexity Analysis **23**

E.1 MCTS Worst Case Complexity Analysis 23

E.2 Simple PMCTS Worst Case Complexity 24

E.3 PMCTS Worst Case Complexity 24

F MCTS and Parallelization: A Brief Survey **25**

F.1 Synchronous vs. asynchronous parallelism 25

F.2 Parallel MCTS: A brief survey 25

G Additional Results **26**

H Implementation Details **28**

H.1 The search policies PUCT and UCT 28

H.2 Improved Policy, Value and Action Selection - A Brief Survey of Design Choices . 28

H.3 The Virtual Losses and Virtual Means Heuristics 30

H.4 Sequential Halving with P/MCTS 30

H.5 Baselines 30

I Experimental Details **31**

I.1 Environments and Scaling Experiments Across Domains 31

I.2 Opening Book 31

I.3 Acting in the Environment 31

I.4 Evaluation Metrics 32

I.5 Discrete vs. Continuous Action Spaces 32

I.6 Runtime Experiments 32
I.7 Policy Improvement Experiment 32
I.8 LLM inference scaling on Sokoban 32

J Hyperparameters **33**
J.1 Hyperparameter tuning 33
J.2 Network architectures 33
J.3 Hyperparameters 33

A Acronym and Symbols List

Variable	Meaning
γ	discount
β_i	Q-value scale at iteration i (inverse-temperature)
s_t	state at step t
τ_t^n	trajectory at step t for particle n
T	last step
$\pi_\theta(s_t)$	prior policy network at state s_t
$v_\phi(s_t)$	value network prediction at state s_t
$\pi'(a_t\|s_t)$	target improved-policy for action a_t at state s_t
$\pi_i(a_t\|s_t)$	selection policy at iteration i for action a_t at state s_t
$\hat{\pi}_i(a_t\|s_t)$	sampling (e.g. *proposal*) policy at state s_t and iteration i
$M_i(s_t)$	visit count (or effective visit count in PMCTS) in the tree for state s_t at iteration i
$v_i(s_t)$	value estimate in the tree for state s_t at iteration i
$v(s_{T+1})$	value prediction at leaf s_{T+1} in the tree during expansion
$q_i(s_t, a_t)$	Q-value estimate in the tree for state s_t and action a_t at iteration i
$N(s_t)$	number of particles passed through state s_t
$ESS(s_t)$	effective sample size for particles that passed through state s_t
w_t^n	weight of particle n at step t
$\bar{w}_T^n(s_t)$	weight of particle n at step T normalized for state s_t
$\nu^n(s_t)$	value update at state s_t by particle n
$\nu(s_t)$	value update at state s_t across all particles
$\eta(s_t)$	possibly state-dependent temperature

B Pseudocode

B.1 Simple PMCTS

Algorithm 1 Simple PMCTS

Require: Number of particles N, Number of iterations M, Root state s_1, Environment model $\mathcal{M}$, DNNs π_θ, v_ϕ.

1: Initialize a tree with capacity for $NM + 1$ nodes.
2: Initialize root using predictions $\pi_\theta(s_1)$ and $v_\phi(s_1)$.
3: **for** $i \leftarrow 1$ to M **do**
4: Sample $\{\tau_T^n = (s_1, a_1^n, ..., s_T^n, a_T^n)\}_{n=1}^N$ from $\pi_i, \mathcal{M}$ independently in parallel. ▷ Selection.
5: Expand and add to the tree up to N leaves in parallel: ▷ Expansion.

$$\left\{s_{T+1}^n \sim P(s_T^n, a_T^n),\, r(s_T^n, a_T^n) = \mathbb{E}[R(s_T^n, a_T^n)],\, v(s_{T+1}^n),\, \pi_\theta(s_{T+1}^n)\right\}_{n=1}^N.$$

6: Compute the value update $\nu_i(s_t)$ (Equation 7, left) $s_t \in \tau_T^n, \forall t, \forall n$: ▷ Backpropagation.

$$\nu_i(s_t) = \frac{1}{N(s_t)} \sum_{n=1}^{N} \mathbb{1}_{s_t \in \tau_T^n} \nu^n(s_t),$$

7: Update the tree (Equation 7, right) $s_t \in \tau_T^n, \forall t, \forall n$

$$v_{i+1}(s_t) = v_i(s_t) + \frac{(\nu_i(s_t) - v_i(s_t))N(s_t)}{M_i(s_t) + N(s_t)}, \quad M_{i+1}(s_t) = M_i(s_t) + N(s_t).$$

8: **end for**
9: **return** : ▷ See Appendix H.2, Equation 50

$$\pi_{search}(a|s_1) = \mathcal{I}(\pi_\theta, q_M), \quad V_{search}(s_1) = \sum_{a \in \mathcal{A}} \bar{\pi}(a|s_1) q_M(s_1, a), \quad a \sim \bar{\pi}(s_1).$$

$$\bar{\pi}(a|s_1) \propto \begin{cases} \pi_{search}(a|s_1) & \text{if: } M(s_1, a) > 0 \\ 0 & \text{if: } M(s_1, a) = 0 \end{cases}$$

B.2 PMCTS

For added clarity, we include a brief summary of PMCTS in natural language in addition and following the pseudocode.

Algorithm 2 PMCTS

Require: Number of particles N, Number of iterations M, Root state s_1, Environment model $\mathcal{M}$, DNNs π_θ, v_ϕ.

Initialize a tree with capacity for $NM + 1$ nodes.
Initialize root using predictions $\pi_\theta(s_1)$ and $v_\phi(s_1)$.
for $i \leftarrow 1$ to M **do**
  for $t \leftarrow 1$ to T for $n \in N$ particles independently in parallel **do** ▷ Weighted Selection
    Sample and append to trajectory τ_{t-1}: ▷ *SMC's Sampling*

$$a_t^n \sim \hat{\pi}(a_t^n|s_t^n), \quad s_{t+1}^n \sim P(\cdot|s_t^n, a_t^n),$$

    Update particle weights (Equation 8): ▷ *SMC's Correction*

$$w_t^n = w_{t-1}^n \frac{\pi_i(a_t^n \mid s_t^n)}{\hat{\pi}_i(a_t^n \mid s_t^n)},$$

  end for
  Expand and add to the tree N leaves in parallel: ▷ Expansion.

$$\left\{s_{T+1}^n \sim P(s_T^n, a_T^n),\, r(s_T^n, a_T^n) = \mathbb{E}[R(s_T^n, a_T^n)],\, v(s_{T+1}^n),\, \pi_\theta(s_{T+1}^n)\right\}_{n=1}^N.$$

  Compute the updated values and policy of the parents of the leaves:

$$\pi_{i+1}(a_T|s_T) \propto \pi_\theta(a_T|s_T) \exp(\beta_{i+1}(s_T) q_{i+1}(s_T^n, a_T^n)).$$

  Update weights w_T^n with respect to $\pi_{i+1}(s_T)$ (Equation 11): ▷ Retrospective reweighting

$$w_T^n \leftarrow w_T^n \frac{\pi_{i+1}(a_T^n \mid s_T^n)}{\pi_i(a_T^n \mid s_T^n)}$$

  Merge duplicates, preserving weight $w_T^n(s_{T+1}) = \sum_{m=1}^N w_T^m \mathbb{1}_{s_{T+1}=s_{T+1}^m}$ per unique trajectory and setting duplicates to zero.
  Normalize weights with respect to each other at all nodes s_t:

$$\bar{w}_T^n(s_t) = \frac{w_T^n \mathbb{1}_{s_t \in \tau_T^n}}{\sum_{m=1}^N w_T^m \mathbb{1}_{s_t \in \tau_T^m}}.$$

  Compute $\nu_i(s_t)$ (Equation 9, left) and the ESS $s_t \in \tau_T^n, \forall t, \forall n$:▷ Weighted Backpropagation.

$$\nu_i(s_t) = \sum_{n=1}^N \bar{w}_T^n(s_t) \mathbb{1}_{s_t \in \tau_T^n} \nu^n(s_t), \quad ESS(s_t) = \frac{1}{\sum_{n=1}^N (\bar{w}_T^n(s_t))^2}$$

  Update the tree (Equation 10) $\forall s_t \in \tau_T^n, \forall t, \forall n$:

$$v_{i+1}(s_t) = v_i(s_t) + \frac{(\nu_i(s_t) - v_i(s_t))\text{ESS}(s_t)}{M_i(s_t) + \text{ESS}(s_t)}, \quad M_{i+1}(s_t) = M_i(s_t) + \text{ESS}(s_t).$$

end for
return : ▷ See Appendix H.2, Equation 50

$$\pi_{search}(a|s_1) = \mathcal{I}(\pi_\theta, q_M), \quad V_{search}(s_1) = \sum_{a \in \mathcal{A}} \bar{\pi}(a|s_1) q_M(s_1, a), \quad a \sim \bar{\pi}(s_1).$$

$$\bar{\pi}(a|s_1) \propto \begin{cases} \pi_{search}(a|s_1) & \text{if:} \quad M(s_1, a) > 0 \\ 0 & \text{if:} \quad M(s_1, a) = 0 \end{cases}$$

PMCTS summarized: At each parallel MCTS iteration i, N particles sample trajectories independently in parallel from a higher-entropy stochastic-selection policy $\hat{\pi}_i$ (Equation 8, right). At each step during selection, the particles' weight are updated with SIS, maintaining a principled estimate of the distribution over trajectories with respect to π_i (Equation 8, left). When all particles arrive at a leaf, N the leaves are expanded in batch parallel, as in Simple PMCTS. Retrospective reweighting is then applied, updating the particles' weights to estimate the distribution with respect to π_{i+1} rather than π_i (Equation 11). Deduplication is applied, removing duplicate trajectories while maintaining the correct importance weight per unique trajectory. Backpropagation is applied, computing the returns ν_i and the ESS and using them to update v_{i+1} (Equations 9 and 10 respectively). At this point, the iteration is complete, and the algorithm proceeds to the next iteration.

C Derivations

C.1 Derivation of the numerically stable weighted average

$$v_{i+1}(s_t) = \frac{\nu(s_t)N_i(s_t) + v_i(s_t)M_i(s_t)}{M_i(s_t) + N(s_t)} \tag{12}$$

$$= \frac{\nu(s_t)N_i(s_t) + v_i(s_t)M_i(s_t) + v_i(s_t)N_i(s_t) - v_i(s_t)N_i(s_t)}{M_i(s_t) + N(s_t)} \tag{13}$$

$$= \frac{v_i(s_t)M_i(s_t) + v_i(s_t)N_i(s_t)}{M_i(s_t) + N_i(s_t)} + \frac{\nu(s_t)N_i(s_t) - v_i(s_t)N_i(s_t)}{M_i(s_t) + N_i(s_t)} \tag{14}$$

$$= v_i(s_t) + \frac{(\nu(s_t) - v_i(s_t))N_i(s_t)}{M_i(s_t) + N_i(s_t)}. \tag{15}$$

C.2 Derivation of the particle-based backpropagation step in PMCTS

C.2.1 Derivation of the variance estimator

We analyze the variance of the PMCTS estimator and construct an estimator of it by considering the set of N particles contributing to the update of node s_t. The estimator is defined as the weighted sum:

$$\mathbb{V}[\nu(s_t)] = \mathbb{V}\left[\sum_{n=1}^{N} \bar{w}_t^n \nu^n(s_t)\right] \tag{16}$$

We assume that we have merged all duplicate particles into one particle with the total weight and all trajectories are unique. While particles in SMC are inherently correlated due to their shared ancestry and the weight normalization constraint $\sum \bar{w}_t^n = 1$, a standard heuristic in the analysis of particle filters is to treat unique trajectories as approximately uncorrelated [26]. Under this assumption, and by treating the weights as approximately constant relative to the variance of the returns, we obtain:

$$\mathbb{V}[\nu(s_t)] \approx \sum_{n=1}^{N} \mathbb{V}[\bar{w}_t^n \nu^n(s_t)] \approx \sum_{n=1}^{N} (\bar{w}_t^n)^2 \mathbb{V}[\nu^n(s_t)] \tag{17}$$

This formulation mirrors the derivation of the Effective Sample Size (ESS) [26], where the variance of a weighted importance sampling estimator is shown to scale with $\sum(\bar{w}_t^n)^2$. The ESS approximation is widely used as a proxy for estimator quality when the true covariance structure is intractable.

Assuming the variance of the return for any single trajectory is bounded by $\mathbb{V}[\nu^n(s_t)] \leq \sigma^2$, we arrive at the following practical bound for the estimator variance:

$$\mathbb{V}[\nu(s_t)] \approx \sum_{n=1}^{N} (\bar{w}_t^n)^2 \mathbb{V}[\nu^n(s_t)] \leq \sigma^2 \sum_{n=1}^{N} (\bar{w}_t^n)^2 = \sigma^2 \frac{1}{ESS(s_t)} \tag{18}$$

This suggests that the variance of the node update is approximately proportional to the sum of the squared weights, $\sum_{n=1}^{N} (\bar{w}_t^n)^2$, allowing us to use this term as a scaling factor for uncertainty in the tree search.

C.2.2 Derivation of the node value update

The weight of the update to the variance-minimizing unbiased estimator of the mean is the inverse variance. Assuming that the variance of all trajectories is bounded by $\sum_{n=1}^{N}(\bar{w}_t^n)^2\sigma^2$, we can divide all updates by σ^2 to arrive at the variance bound being $\sum_{n=1}^{N}(\bar{w}_t^n)^2$. The inverse variance is then $\frac{1}{\sum_{n=1}^{N}(\bar{w}_t^n)^2} = ESS(s_t)$ (the *effective sample size*, see [26]).

We can now derive the update:

$$v_{i+1}(s_t) = \frac{\nu(s_t)\text{ESS}(s_t) + v_i(s_t)M_i(s_t)}{M_i(s_t) + \text{ESS}(s_t)} \tag{19}$$

$$= \frac{\nu(s_t)\text{ESS}(s_t) + v_i(s_t)M_i(s_t) + v_i(s_t)\text{ESS}(s_t) - v_i(s_t)\text{ESS}(s_t)}{M_i(s_t) + \text{ESS}(s_t)} \tag{20}$$

$$= \frac{v_i(s_t)M_i(s_t) + v_i(s_t)\text{ESS}(s_t)}{M_i(s_t) + \text{ESS}(s_t)} + \frac{\nu(s_t)\text{ESS}(s_t) - v_i(s_t)\text{ESS}(s_t)}{M_i(s_t) + \text{ESS}(s_t)} \tag{21}$$

$$= v_i(s_t) + \frac{(\nu(s_t) - v_i(s_t))\text{ESS}(s_t)}{M_i(s_t) + \text{ESS}(s_t)}. \tag{22}$$

C.3 Approximating expectations under the target in Sequential Monte Carlo

There are multiple approaches to approximating expectations under the target using the particles $\{\tau_t^n, w_t^n\}_{n=1}^N$ of SMC, for some example function $\mathcal{R}$ (in RL terms, we can think of $\mathcal{R}(\tau_T^n)$ as the return along the trajectory).

The standard approach is to use the *self-normalizing* estimator:

$$\frac{1}{\sum_{n=1}^{N} w_t^n}\sum_{n=1}^{N} w_t^n \mathcal{R}(\tau_t^n) \approx \mathbb{E}_{\pi'}[R(\tau_t^n)] = V^{\pi'}(s_1). \tag{23}$$

It normalizes with respect to the sum of the *weights* of the particles $\sum_{n=1}^{N} w_t^n$. It is the more common of the options, as it allows for SIS even with access only to the probability mass / density functions $\pi'(a|s), \pi_\theta(a|s)$ up to a proportionality constant. This estimator however is biased (although consistent), introducing bias of order $\mathcal{O}(1/N)$ [45], because the ratio of two unbiased estimators: $\sum_{n=1}^{N} w_t^n \mathcal{R}(\tau_t^n) / \sum_{n=1}^{N} w_t^n$, is not itself unbiased [26].

However, when we have access to the full densities / masses $\pi'(a|s), \pi_\theta(a|s)$, we can also use the estimator (Equation 5):

$$\frac{1}{N}\sum_{n=1}^{N} w_t^n \mathcal{R}(\tau_t^n) \approx \mathbb{E}_{\pi'}[R(\tau_t^n)] = V^{\pi'}(s_1). \tag{24}$$

This is an *unbiased* estimator of expectations under the function $\mathcal{R}$.

In practice, the first estimator (Equation 23) is much lower variance, despite being biased, and we use it in the practical algorithm.

C.4 Unbiased estimates of the value at any depth

The unnormalized estimator provides unbiased estimates of the expectation at the root s_1. However, we cannot simply use the same weights for any node under the root (the way we could for the self-normalizing estimator) and expect unbiased expectations, because the weights will be proportional to the probability of reaching those nodes. As a result, unlikely trajectories will have value estimates close to zero (because their weights will be close to zero) and using them directly to update the tree will effectively set all the values closer to zero.

This can be addressed however, by maintaining an array of weights per particle n, which save a weight per depth t. At each depth t the new-weight to depth t is initialized with 1.0, and is updated in the same manner as all other weights at all depths. As a result, a different weight is available per particle per depth to compute the expectation at each depth (i.e. each node along the trajectory). This

can effectively be thought of as "restarting" the SMC process at each depth guaranteeing that its properties apply for each depth and not just the root.

In practice however, the self normalizing estimator is the popular estimator which will be used in practice in most practical settings and much-reduced bias versions of it exist [45].

D Formal Theoretical Analysis

D.1 MCTS as a principled policy improvement operator

We begin with the formal Theorem 1, and proceed with its proof.

Theorem 4 (MCTS as a Principled Policy Improvement Operator (Formal))**.** *Given selection policy π_i (Equation 2), a deterministic true model $\mathcal{M}$, evaluations $v(s_{T+1})$ of leaves s_{T+1} sampled independently from a distribution with mean $V^{\pi_\theta}(s_{T+1})$ and variance $\leq \sigma^2$ (i.e. the relaxed assumptions, which contain the classical), π_θ having support over the entire domain $\mathcal{A}$ and in expectation with respect to backpropagation: $\nu_i(s_t) \leftarrow \mathbb{E}\big[\mathcal{R}(\tau_{t,T}) + V^{\pi_\theta}(s_{T+1})\big]$, MCTS satisfies Policy Improvement:*

$$\forall i \geq 1: \quad V^{\pi_i}(s_1) \geq V^{\pi_\theta}(s_1). \tag{25}$$

Proof.

Preliminaries: Let π_θ be the prior policy with support over the entire domain (required for $\mathcal{I}_{GMZ}$ to be defined) and $v(s)$ be an unbiased value estimator such that $\mathbb{E}[v(s)] = V^{\pi_\theta}(s)$ with finite variance (e.g. rollouts).

We define the MCTS iteration sequence for $i = 1, \ldots, M$. At each iteration i, a selection policy π_i is determined using q_i and π_θ as follows:

$$\pi_i(a|s) = \mathcal{I}_{GMZ}(\pi_\theta, q_i)(a|s) \propto \pi_\theta(a|s) \exp(\beta_i(s) q_i(s,a)). \tag{26}$$

We define the mixture policy π^i as the policy which selects $\pi_1, \ldots, \pi_i$ with equal probability at state s, and follows the selected policy from s.

Lemma 1 (The expected backpropagation)**.** *The expected backpropagation $\mathbb{E}[\nu_i(s_t)]$ at any node s_t in the search tree is the value of the selection policy $V^{\pi_i}(s_t)$.*

Proof Lemma 1. Taking the expectation with respect to all random variables:

$$\mathbb{E}[\nu_i(s_t)] = \mathbb{E}\left[\sum_{j=t}^{T} \gamma^{j-t} r(s_j, a_j) + \gamma^{T+1-t} v(s_{T+1})\right] \tag{27}$$

$$= \mathbb{E}\left[\sum_{j=t}^{T} \gamma^{j-t} r(s_j, a_j) + \gamma^{T+1-t} \mathbb{E}[v(s_{T+1})]\right] \tag{28}$$

$$= \mathbb{E}\left[\sum_{j=t}^{T} \gamma^{j-t} r(s_j, a_j) + \gamma^{T+1-t} V^{\pi_\theta}(s_{T+1})\right] \tag{29}$$

$$= V^{\pi_i}(s_t) \tag{30}$$

Since π_i is followed until the leaf s_{T+1}, and π_θ is implicitly followed thereafter (via the value head v), this expression is precisely the value function of a policy that follows π_i for T steps and then switches to π_θ. In the context of MCTS, this defines the value of the selection policy π_i. □

Lemma 2 (Value of the Mixture Policy)**.** *The state-action value $q_{i+1}(s_t, a)$ with expected evaluations $\mathbb{E}[\nu_i(s_t)]$ at every step is the Q-value of the mixture policy π^i: $q_{i+1}(s_t, a) = Q^{\pi^i}(s_t, a)$.*

Proof Lemma 2. Let iterations i enumerate all iterations through node s_t and action a without loss of generality. By the definition of the MCTS node evaluations and Lemma 1:

$$q_{i+1}(s_t, a) = \frac{1}{i}\sum_{j=1}^{i} Q^{\pi_j}(s_t, a) = \sum_{j=1}^{i} \frac{1}{i} Q^{\pi_j}(s_t, a) \tag{31}$$

By the linearity of the Q-function with respect to the policy distribution in an MDP:

$$\sum_{j=1}^{i} \frac{1}{i} Q^{\pi_j}(s_t, a) = Q^{\pi^i}(s_t, a). \tag{32}$$

where $Q^{\pi^i}(s_t, a)$ is the value of the policy which picks $\pi_1(s_t), \ldots, \pi_i(s_t)$ with equal probability and acts with that policy to termination. □

Lemma 3. *A mixture of improved policies is an improved policy.*

Proof Lemma 3. Lemma 3 is proven by [34] in their Lemma 1. □

D.1.1 Proof by induction for Theorem 4:

Using Lemmas 1,2,3 we will prove the following by induction: *If evaluations $v(s_{T+1})$ of leaves s_{T+1} are sampled i.i.d. from a distribution with mean $V^{\pi_\theta}(s_{T+1})$ and variance $\leq \sigma^2$, $\pi_i = \mathcal{I}_{GMZ}(\pi_\theta, q_i)$ and π_θ has support over the entire domain then in expectation with respect to search MCTS satisfies policy improvement:*

$$\forall i \geq 1 : \quad V^{\pi_i}(s_1) \geq V^{\pi_\theta}(s_1). \tag{33}$$

Inductive Hypothesis For all i: $V^{\pi_i}(s_t) \geq V^{\pi_\theta}(s_t),\ q_i(s_t, a) \geq Q^{\pi_\theta}(s_t, a)$.

Base case $i = 1$ At the first iteration $i = 1$ through any node s_t (which is the expansion s_t, a_t) we have $\pi_1(s_t) = \pi_\theta(s_t)$. By Lemma 1 we have $q_i(s_t, a) = Q^{\pi_\theta}(s_t, a)$. QED base case.

Inductive step We need to show that $V^{\pi^{i+1}}(s_t) \geq V^{\pi_\theta}(s_t)$ and $q_{i+1}(s_t, a) \geq Q^{\pi_\theta}(s_t, a)$, given $V^{\pi_i}(s_1) \geq V^{\pi_\theta}(s_1)$ and $q_i(s_t, a) \geq Q^{\pi_\theta}(s_t, a)$ the inductive hypothesis.

From Lemma 2 $q_{i+1}(s_t, a) = Q^{\pi^i}(s_t, a)$, the value of the mixture policy π^i. From the inductive hypothesis, all $\pi_j, j \leq i$ are improvements over π_θ. By Lemma 3 their mixture π^i is also an improved policy: $V^{\pi^i}(s_1) \geq V^{\pi_\theta}(s_1)$. Therefore, the value $q_{i+1}(s_t, a) = Q^{\pi^i}(s_t, a) \geq Q^{\pi_\theta}(s_t, a)$ is the value of an improved policy at any state s_t and action a.

The selection policy π_{i+1} is defined with:

$$\pi_{i+1}(a|s_t) = \mathcal{I}_{GMZ}(\pi_\theta, q_{i+1})(a|s), \tag{34}$$

which satisfies policy improvement over π_θ by definition. Therefore, we have:

$$V^{\pi_{i+1}}(s_1) \geq V^{\pi_\theta}(s_1). \tag{35}$$

QED Inductive step.

Conclusion By induction, every selection policy π_{i+1} generated by the MCTS operator with respect to expected evaluations $\mathbb{E}[\nu_i(s_t)]$ at every step is a policy improvement over the prior π_θ at any state s_t. By generality, this applies to the root state s_1 as well.

Therefore we have, under the assumptions stated in the Theorem, that:

$$\forall i \geq 1 : V^{\pi_i}(s_1) \geq V^{\pi_\theta}(s_1). \tag{36}$$

□

D.2 Simple PMCTS is a principled policy improvement operator

We begin with the formal Theorem 2, and proceed with its proof.

Theorem 5 (Simple PMCTS is a Principled Policy Improvement Operator (Formal))**.** *Given selection policy π_i (Equation 2), a deterministic true model $\mathcal{M}$, evaluations $v(s_{T+1})$ of leaves s_{T+1} sampled i.i.d. from a distribution with mean $V^{\pi_\theta}(s_{T+1})$ and variance $\leq \sigma^2$ (i.e. the classical assumptions), π_θ having support over the entire domain $\mathcal{A}$, and when $N \to \infty$, Simple PMCTS satisfies Policy Improvement:*

$$\forall i \geq 1: \quad V^{\pi_i}(s_1) \geq V^{\pi_\theta}(s_1), \quad \textit{a.s.} \tag{37}$$

Proof. Consider iteration i and condition on the search history prior to iteration i, which fixes the selection policy π_i. Since π_i has support over all actions supported by π_θ, every reachable node s_t is visited with positive probability. Hence, as $N \to \infty$, the number of particles reaching every reachable node satisfies

$$N_i(s_t) \to \infty \qquad \text{a.s.}$$

At any such node s_t, the returns used for backpropagation are conditionally i.i.d., with mean

$$\mathbb{E}[\mathcal{R}(\tau_{t,T}) + V^{\pi_\theta}(s_{T+1}) \mid s_t] = V^{\pi_i}(s_t),$$

and finite variance by assumption. Thus, by the strong law of large numbers, the empirical backpropagation estimate satisfies

$$\nu_i(s_t) \xrightarrow[N\to\infty]{\text{a.s.}} V^{\pi_i}(s_t).$$

Therefore, in the limit $N \to \infty$, the backpropagation of Simple PMCTS is exactly the expected backpropagation of Theorem 1, almost surely.

Let E_i denote the probability-one event on which this convergence holds for every node in iteration i. Since the set of iterations and the set of nodes are countable,

$$\Pr\left(\bigcap_{i=1}^{\infty} E_i\right) = 1.$$

On this event, Simple PMCTS coincides with the expected-backpropagation MCTS of Theorem 1 at every iteration. Consequently, by Theorem 1,

$$\forall i \geq 1: \qquad V^{\pi_i}(s_1) \geq V^{\pi_\theta}(s_1).$$

Hence Simple PMCTS guarantees policy improvement at every iteration almost surely. □

D.3 PMCTS is a principled policy improvement operator

We begin with the formal Theorem 3, and proceed with its proof.

Theorem 6 (PMCTS is a Principled Policy Improvement Operator (Formal))**.** *Given selection policy π_i (Equation 2), a deterministic true model $\mathcal{M}$, evaluations $v(s_{T+1})$ of leaves s_{T+1} sampled i.i.d. from a distribution with mean $V^{\pi_\theta}(s_{T+1})$ and variance $\leq \sigma^2$ (i.e. the classical assumptions), π_θ having support over the entire domain $\mathcal{A}$ and when $N \to \infty$, PMCTS satisfies Policy Improvement:*

$$\forall i \geq 1: \quad V^{\pi_i}(s_1) \geq V^{\pi_\theta}(s_1). \quad \textit{a.s.} \tag{38}$$

Proof. We show that in the limit $N \to \infty$, each of the mechanisms introduced by PMCTS preserves the expected-backpropagation limit of MCTS.

Consider iteration i and condition on the search history prior to iteration i. This fixes the selection policy π_i and hence the proposal policy $\hat{\pi}_i$. Since π_θ has full support over $\mathcal{A}$ and π_i and $\hat{\pi}_i$ are constructed from π_θ by strictly positive transformations, π_i and $\hat{\pi}_i$ have the same support. Consequently, every trajectory with positive probability under π_i also has positive probability under $\hat{\pi}_i$.

Under the classical assumptions, the particles are sampled independently from $\hat{\pi}_i$, and their sequential importance weights are

$$w_t^n = w_{t-1}^n \frac{\pi_i(a_t^n \mid s_t^n)}{\hat{\pi}_i(a_t^n \mid s_t^n)}.$$

Thus, at any node s_t, the PMCTS backup $\nu_i(s_t)$ is a self-normalized importance-sampling estimator of the expectation under the target policy π_i. More generally, after retrospective reweighting, the same estimator is a self-normalized importance-sampling estimator under the updated target policy used by the subsequent MCTS update.

Since the importance weights have finite expectation and the backpropagated returns have finite first moment, the strong law of large numbers for self-normalized importance sampling gives

$$\nu_i(s_t) \xrightarrow[N\to\infty]{\text{a.s.}} V^{\pi_i}(s_t),$$

with the corresponding target policy after retrospective reweighting. The result also holds at every depth because PMCTS maintains depth-specific importance weights, so that the self-normalized estimator is applied to the particles contributing to each node.

Deduplication does not alter this limit. It only aggregates particles corresponding to identical trajectories by summing their importance weights. Hence the weighted empirical distribution, and therefore the self-normalized estimator $\nu_i(s_t)$, is unchanged.

After retrospective reweighting, the estimator converges to the value of the corresponding updated selection policy π_{i+1}. However, since this policy remains an improved policy which maintains the same properties as π_i, we overload the notation and set π_i as the policy in Equation 2 with respect to q_i, β_i at all nodes that have not been expanded, and Equation 2 with respect to q_{i+1}, β_{i+1} at expanded nodes, without influencing the convergence or the recursion.

The ESS-based update also preserves the expected-backpropagation limit. Since the SMC estimator is consistent,

$$\mathrm{Var}[\nu_i(s_t)] \to 0, \qquad \frac{1}{\mathrm{ESS}(s_t)} \to 0 \quad \text{a.s.}$$

and hence $\mathrm{ESS}(s_t) \to \infty$ almost surely. Therefore, in the limit $N \to \infty$, the update

$$v_{i+1}(s_t) = v_i(s_t) + \frac{\mathrm{ESS}(s_t)}{M_i(s_t) + \mathrm{ESS}(s_t)} \left(\nu_i(s_t) - v_i(s_t)\right)$$

becomes a convex combination of the previous tree value and the exact expected-backpropagation value $\nu_i(s_t)$. The previous tree value is, by induction, the value of a mixture of policies that satisfy policy improvement, while $\nu_i(s_t)$ is the value of the newly improved selection policy. Thus the resulting $v_{i+1}(s_t)$ retains the same policy-improvement property as in Theorem 4.

Finally, consider the randomness across MCTS iterations. Let E_i denote the probability-one event on which the above convergence holds for every node involved in iteration i. Since the set of iterations and the set of nodes are countable,

$$\Pr\left(\bigcap_{i=1}^{\infty} E_i\right) = 1.$$

On this event, PMCTS coincides with the expected-backpropagation MCTS in the limit $N \to \infty$, up to the convex averaging of values from previous improved policies. The proof of Theorem 4 therefore applies by induction over iterations. Hence,

$$\forall i \geq 1: \qquad V^{\pi_i}(s_1) \geq V^{\pi_\theta}(s_1), \qquad \text{a.s.}$$

Thus PMCTS guarantees policy improvement at every iteration almost surely in the limit $N \to \infty$. □

E Worst-Case Complexity Analysis

In this section we analyze the computation complexity of Simple PMCTS (Algorithm 1) and PMCTS (Algorithm 2). We begin with a brief complexity analysis of baseline MCTS, for reference.

E.1 MCTS Worst Case Complexity Analysis

MCTS is given a search budget of M *simulations*, i.e. iterations of selection, expansion, and backpropagation. Expansion takes a constant amount of time, and both selection and backpropagation

scale linearly with the depth d_i of the newly expanded leaf. $d_i \leq M$ since in each iteration, the depth of the tree increases by at most one. Thus we can bound the worst-case runtime complexity by $\mathcal{O}(M^2)$.

Each expansion step adds a new node to the tree, and the rest of the operations (selection and backup) use a constant amount of space, so in total the space complexity of MCTS is $\mathcal{O}(M)$.

E.2 Simple PMCTS Worst Case Complexity

We analyze the worst-case complexity of Simple PMCTS (and next PMCTS) with respect to the search budget of M *simulations* (i.e. iterations) and number of particles N.

Selection: Each particle can do selection completely independently, so the time taken again scales linearly in the maximum depth across particles d_i. Even though we may add up to N nodes in each iteration, the depth still only increases by at most one, so we have $d_i \leq M$ again.

Expansion: Expansion is done as a batch. We compute the value predictions, new prior policies, and add up to N nodes to the tree. This can be done in constant time by having every particle create the new node in a predetermined spot based on its index. To connect the parent nodes to the new children, each particle tries to assign its child index to the parent, and in the case of contention, one arbitrarily wins out. Each particle then checks what was written and reassigns itself to the child node that won.

Backpropagation: Each particle can independently calculate its contribution to each node by tracing its trajectory in the tree. This will take a linear time in d_i. The contributions can then be combined for each state in parallel, either with a linear scan $\mathcal{O}(N)$ or tree reduction $\mathcal{O}(\log(N))$ [46]. Thus the time for backpropagation scales as $\mathcal{O}(d_i + \log(N))$.

Total: In each iteration we take $\mathcal{O}(d_i)$ for selection, $\mathcal{O}(1)$ time for expansion, and $\mathcal{O}(d_i + \log(N))$ for backpropagation. Over M iterations this amounts to $\mathcal{O}(M(M + \log(N))) = \mathcal{O}(M^2 + M\log(N))$.

Space: Since each iteration can add at most N new nodes, the space required for the search tree is $O(MN)$. Additional space is required during backpropagation to keep track of the contribution of each particle at each affected node: There are N particles and each affected at most d_i nodes, so $\mathcal{O}(d_i N) \leq \mathcal{O}(MN)$ space.

E.3 PMCTS Worst Case Complexity

Selection: Selection is much the same as in Simple PMCTS. The few differences are that we must now compute a sampling policy $\hat{\pi}_i$ to sample from, and the particle weights must be updated. The weight update is constant-time per step, and when using a temperature-based sampling policy as in this work, $\hat{\pi}_i$ can also be computed in constant time (independently by each particle). This gives the same linear time $\mathcal{O}(d_i) \leq \mathcal{O}(M)$ for selection.

Expansion remains identical to Simple PMCTS, so $\mathcal{O}(1)$.

Backpropagation: Most of PMCTS's features come into play in this step. We compute the unweighted returns contributed by each particle in parallel in $O(d_i)$ time. *Retrospective reweighing* can be done in parallel for each particle in constant time. De-duplication of particles can be done using *reduce-by-key* in $\mathcal{O}(\log(N))$ time [47] (i.e. by using a parallel sort, detecting boundaries, and then tree-reducing). Computing ESS can be done for each affected state in parallel in time $\mathcal{O}(\log(N))$ using tree reduction for sums over particle weights, and parallelism across particles for operations such as squaring or dividing weights. Calculating the weighted return similarly takes $\mathcal{O}(\log(N))$ time with tree reduction and can be done in parallel per state. In total the worst-case time complexity for backup is $\mathcal{O}(d_i + \log(N))$.

Total: Similar to Simple PMCTS, the worst-case complexity of one iteration is $\mathcal{O}(M + \log(N))$ and $\mathcal{O}(M^2 + M\log(N))$ the entire algorithm.

Space: The space required for the search tree is $\mathcal{O}(MN)$. Space required for backpropagation is dominated by the weighted return and ESS calculations which need $\mathcal{O}(d_i N) \leq \mathcal{O}(MN)$ as before to store the contribution of each particle.

F MCTS and Parallelization: A Brief Survey

F.1 Synchronous vs. asynchronous parallelism

A system of parallel processes is called *synchronous* if all processes run using the same clock, i.e., all operations are executed in lockstep (physically or conceptually). It is called *asynchronous* if each process has its own independent clock, i.e. the different processes execute independently, potentially interleaving arbitrarily [48].

Asynchronous parallelization is very popular, as it allows multiple processes to run independently in parallel, and is indeed a popular way to implement virtual-visits based heuristic parallelization of MCTS [49].

In contrast, modern GPU acceleration with Jax [25] enables a form of semantically synchronous parallelization, typically expressed via batching or vectorization. In this setting, computations are defined so that elements of a batch are processed independently but conceptually in lockstep, with no observable ordering or interaction between them. This has popularized synchronous parallelized acceleration of the RL framework, based on “batched" operations, for training as well as search [24, 34, 38]. As a result, it is much easier to benefit from GPU acceleration with Jax with conceptually synchronous parallelization.

F.2 Parallel MCTS: A brief survey

Many approaches for parallelizing MCTS under different practical constraints have been proposed over the years [17, 18, 20, 22, 50–52]. Among them, the three canonical approaches are *root parallelization*, *leaf parallelization* and *tree parallelization* [17, 18].

Root parallelization parallelizes across multiple instances of MCTS. To achieve runtime inference scaling, all instances run from the same root state and aggregate their results once search has concluded. This approach builds search trees of the same size, and in expectation each tree is the same. It also relies heavily on stochasticity in the selection and/or evaluation and thus is not well suited to modern uses of MCTS that rely on fully deterministic selection and evaluation.

A variant of this approach is very effective for scaling up during training however. Instead of running multiple instances of MCTS in the same state in parallel and aggregating their results, the search can be parallelized across N *different* states. The results are then aggregated in a training batch over these states. This approach is the go-to in Jax based implementations of training RL agents with MCTS for policy improvement (e.g. AlphaZero variants) [43].

Leaf parallelization parallelizes the evaluation of the same leaf across N parallel processes. This is only effective when evaluation is stochastic, and thus is not suited for modern algorithms either.

Tree parallelization parallelizes the search with multiple parallel, synchronous or asynchronous, *selection, expansion* and *backpropagation* processes operating on the same tree. The immediate challenge these methods run into are duplicate selection trajectories. To overcome this challenge, the canonical approach in practice is the *virtual visits* family of heuristics. To reduce the probability that different processes attempt to expand the same leaf, the main idea is that when a process arrives at a node during selection, it appends a count of "virtual" visits *before* the expansion and backpropagation steps complete. These visits are taken into account by other processes arriving at the same name, *as if* the node was fully updated with backpropagation.

This approach is very popular in practice, and has been very successful for benefiting from parallel compute during inference time. However, it is not principled, in the sense that there are no guarantees over which policy is evaluated by the tree at each iteration.

Additional approaches have iterated on the notion of virtual visits with *informed*-virtual visits, using smaller DNNs or other forms of cheaper evaluations to direct the search, while spending the cost for expensive evaluations only later or only on specific nodes [53–55].

Prior work analyzed this direction theoretically from the perspective of *how much worse than regular MCTS can the decisions induced by virtual visits be* [21]. In contrast, in this work we develop methods that both do as well as MCTS in expectation (unbiased), and reduce variance in proportion to the number of parallel processes.

Overall, a large number of approaches have investigated using virtual visits in different implementations with different levels of synchronous and asynchronous parallelization [17, 18, 20, 22, 50, 51].

Other approaches for parallelizing MCTS have been proposed. [52] replaces the DNN-learned dynamics model of MuZero [4] with a transformer, to unroll multiple action trajectories in the tree in a batch. This approach is not suited for MCTS in general, only for DNN-based dynamics models, modifies the backpropagation of MCTS, and does not provide any theoretical analysis or guarantees of policy improvement with respect to the modified selection policies.

G Additional Results

Scaling across M in an example environment is investigated in Figure 4.

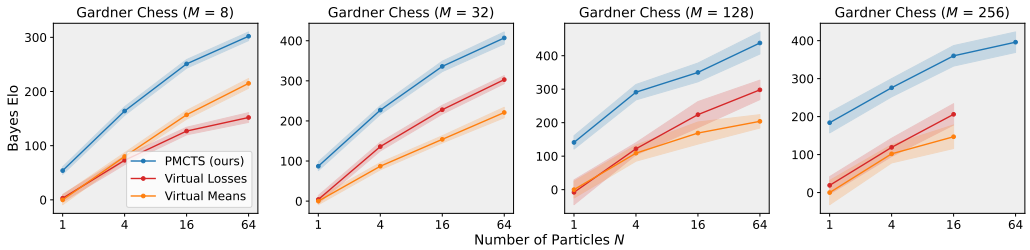


Figure 4: Scaling of parallel MCTS variants with parallel compute (number of particles N) and simulation budgets M in Gardner chess. Bayes Elo with 95% CI.

The individual boardgames results aggregated in Figure 2 are presented in Figure 5.

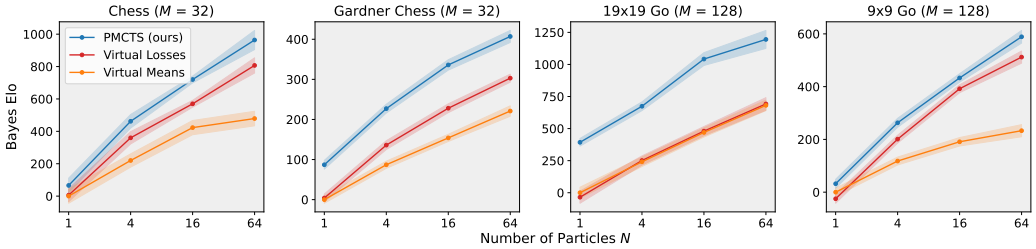


Figure 5: Scaling of tree parallel MCTS variants with parallel compute (number of particles N) in different boardgames. Bayes Elo with 95% CI.

The individual discrete and continuous action results aggregated in Figure 2 are presented in Figure 6.

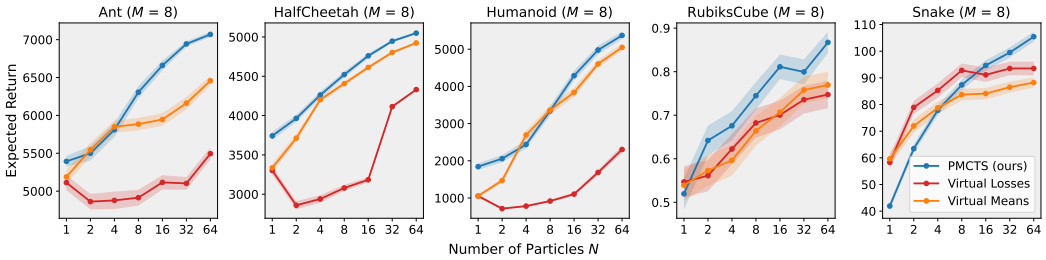


Figure 6: Scaling of parallel MCTS variants with parallel compute (number of particles N) across different environments in the discrete and continuous domains. Mean expected return with 95% Gaussian CI.

A comparison between tree-parallel MCTS with PMCTS and root-parallel MCTS is presented in Figure 7, demonstrating that indeed, root-parallelization is not an effective runtime-scaling approach when DNNs are used for evaluation. For this comparison, we use a modern version of non-parallel MCTS, GumbelMCTS, and run it vectorized across different instances at the same state, which also allows us to compare the performance of PMCTS to a popular version of non-parallel MCTS.

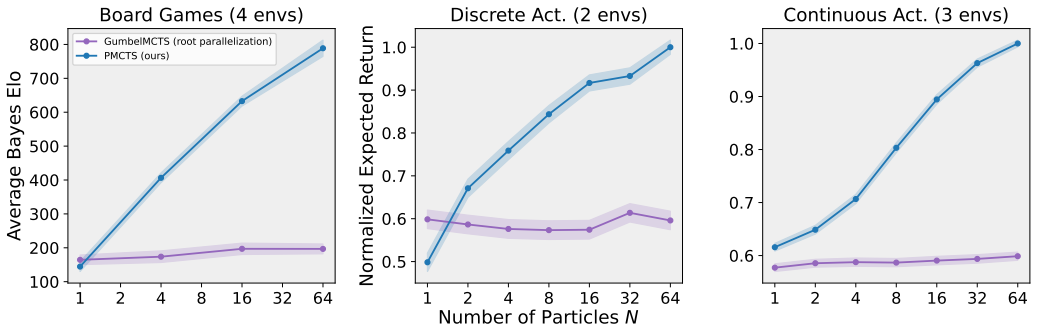


Figure 7: Scaling of tree parallel MCTS with PMCTS vs, root parallel MCTS with GumbelMCTS across boardgames, discrete and continuous environments. Average Bayes Elo or expected return with 95% CI.

In Figure 8 we investigate the effect of different action selection mechanisms across the different baselines. We find that our experiments use the best action selection mechanism per algorithm. *Improved Policy* refers to selecting the action that maximizes the improved policy. *Max. Visits* chooses the action with the most visits. For GumbelMCTS, we compare different ways to aggregate across trees, specifically averaging the improved policy (*Aggregate Policy*), averaging the Q-values (*Aggregate Values*), and choosing the action picked by the most independent searches (*Aggregate Voting*), see Appendices I.3 and H.2 for more detail.

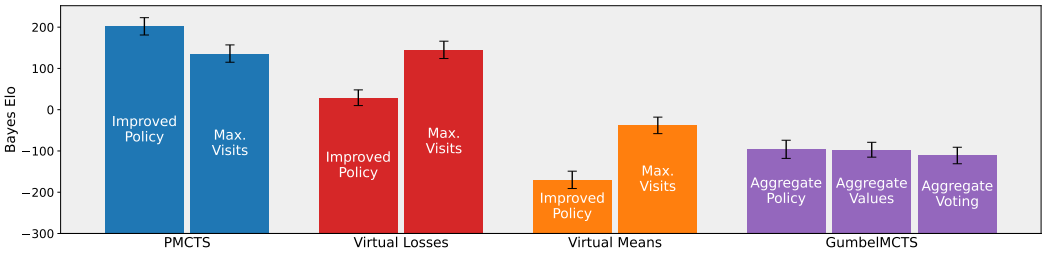


Figure 8: Action selection ablations across the different baselines, in 9x9 Go ($M = 128, N = 16$). Bayes Elo with 95% CI.

We include a comparison to TSMCTS and SMC on Garnder chess and 9x9 Go, demonstrating their challenge to perform well with a larger simulations budgets (and smaller particle budgets, in the case of TSMCTS), in Figure 9.

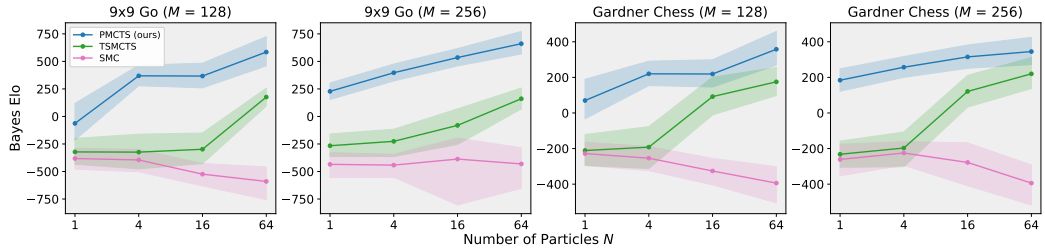


Figure 9: PMCTS vs. TSMCTS and baseline SMC in Gardner chess and 9x9 Go, with larger simluation budgets $M = 128, 256$. Bayes Elo with 95% CI.

To investigate how the temperature η trades off between increased number of unique trajectories and reduced divergence from the target π_i we include search statistics averaged over multiple trees at different states in different games of 9x9 Go with $M = 128, N = 64$ (unless otherwise specified), in Figure 10. On the left, we see that as the temperature increases, the number of unique trajectories increases, as expected. Second from left, we see that as the temperature increases, the divergence between the sampling policy $\hat{\pi}_i$ and the target policy π_i increases, as expected. On the third from the left we see that the temperature parameter that consistently achieves the lowest variance $\mathbb{V}[\nu_i]$, estimated with 1/ESS – the best tradeoff – is $\eta = 1.5$ (marked in blue for clarity). We also see that it achieves a rather-large number of unique trajectories (left), and rather-low KL divergence (second

from left), as expected from the temperature that trades off the best between the two to minimize variance. Finally, we can see that indeed this results in the best performance (right), as expected.

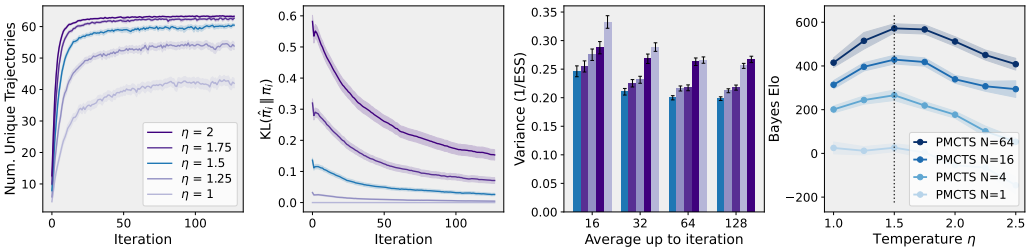


Figure 10: Search statistics for PMCTS on 9x9 Go with $M = 128$. Means and 95% Gaussian or Bayes Elo CIs, where relevant. The temperature that achieves the best tradeoff $\eta = 1.5$, the one that minimizes variance (second from right), is also the one that achieves the best performance (right).

H Implementation Details

H.1 The search policies PUCT and UCT

$$\pi_i^{\text{UCT}}(s) := \arg\max_{a \in A} q_i(s_t, a) + \pi_\theta(a|s_t)\, C \sqrt{\frac{\log M_i(s_t)}{1 + M_i(s_t, a)}}. \tag{39}$$

$$\pi_i^{\text{PUCT}}(s) := \arg\max_{a \in A} q_i(s_t, a) + \pi_\theta(a|s_t)\, C \frac{\sqrt{M_i(s_t)}}{1 + M_i(s_t, a)}. \tag{40}$$

H.2 Improved Policy, Value and Action Selection - A Brief Survey of Design Choices

At the end of search, P/MCTS has q_M and π_M from which it needs to return an action to take in the environment, an improved policy π_{search} to train the policy network π_θ, and value V_{search} to train the value network v_ϕ. This can be (and is) done in multiple different ways, which we discuss below.

MCTS for Heuristic Policy Improvement: In AlphaZero [3], MCTS is used to train the policy. As prior to [32]'s work, MCTS was not theoretically interpretable as a policy improvement operator, it was not clear how to extract a target for the prior policy. [3] chose the visitations distribution $M(s_1, a)$:

$$\pi_{\text{search}}^{\text{classical}}(a|s_1) = \frac{M(s_1, a)}{\sum_{b \in \mathcal{A}} M(s_1, b)} \tag{41}$$

Although this does not guarantee policy improvement [24], it has been extremely successful in practice and paved the way for the great influence MCTS-based agents had on RL.

It is natural to then compute the root value and select an action to take as:

$$V_{\text{search}}(s_1) = \sum_{a \in \mathcal{A}} \pi_{\text{search}}^{\text{classical}}(a|s_1) q_M(s_1, a), \quad a \sim \pi_{\text{search}}^{\text{classical}}(s_1). \tag{42}$$

When we want a deterministic action, we can use select the action with the most visits $M(s_1, a)$.

MCTS as a Policy Improvement Operator [32] showed that MCTS *tracks*, *approximates* and therefore *can be interpreted as* a policy improvement operator. More specifically, their results can be summarized as follows: (I) The popular selection policies UCT and PUCT *converge over iterations* (e.g. *track*) to the solution of an optimization problem of the form:

$$\pi'(s) = \arg\max_{\pi'} \sum_{a \in \mathcal{A}} \pi'(a|s) q_i(s, a) - \lambda_i D(\pi', \pi_\theta). \tag{43}$$

Here, D is some regularization term and λ_M is a function of the visitation counts $M(s, a)$. Further, [32] showed that PUCT corresponds to the the specific regularization term $D(\pi', \pi_\theta) = KL(\pi_\theta, \pi')$, i.e. the reverse KL divergence. We apply the following definition of the $KL(x, y) = \sum_{a \in \mathcal{A}} x(a) \log \frac{x(a)}{y(a)}$.

Next, [32] showed that the solution π' (Equation 43) can be used directly, for three purposes: First, for selection actions in the environment. Second, for training the policy. And third, for driving the search itself, replacing deterministic PUCT with stochastic π'.

The operator $\pi' = \mathcal{I}(\pi_\theta, q_M)$ which solves Equation 43 is not a very popular operator, and requires solving an optimization problem numerically to find the normalization constant of π'. [24] showed that it can be replaced by $\mathcal{I}_{\text{reg}}$ effectively.

Respectively, this suggests the following natural return values:

$$\pi_{\text{search}}(a|s_1) = \mathcal{I}(\pi_\theta, q_M),\ V_{\text{search}}(s_1) = \sum_{a\in\mathcal{A}} \pi_{\text{search}}(a|s_1) q_M(s_1, a),\ a \sim \pi_{\text{search}}(s_1). \tag{44}$$

Sequential Halving and Completed Qs: [24] introduced *Sequential Halving* (SH) to search at the root. SH is a bandit algorithm which optimizes for *simple-regret, best action identification* given *known* search budget M. In contrast, PUCT was designed for *anytime, cumulative-regret, best action identification*. SH can be interpreted in two different ways: First, it is just a budget-allocation algorithm, which assigns different actions different action budgets, without a direct influence on the action values themselves. The one exception to this statement is that in expectation, the values of all actions *increase* with increased search resources (Theorem 1). This was modeled as *drift* by [1]. From the perspective of policy improvement however, irrespective of the resource allocation at the root, the algorithm remains an unbiased estimator of policy improvement.

Second, SH can be viewed as a *best-action selection* algorithm. At the last iteration of SH, it outputs a recommended-best action. [24] used SH to select actions in the environment. However, they found that in order to retain sufficient exploration in board games, the agents benefited from sampling proportional to $\pi_{search}^{classical}$, the policy induced directly by the visitation counts $M(s, a)$.

Additionally, [24] proposed the completed Q values, which predicts the value $q(s, a)$ of unvisited actions $M(s, a) = 0$ with a mixture of the form:

$$q_i(s,a) = \begin{cases} q_i(s,a) & \text{if} M(s,a) > 0 \\ v_{\text{mix}}(s) & \text{if} M(s,a) = 0 \end{cases}, \tag{45}$$

$$v_{\text{mix}}(s) = \frac{1}{1 + \sum_a M(s,a)} \left(v_\phi(s) + \frac{\sum_a M(s,a)}{\sum_{a\in A} \pi_\theta(a|s)} \sum_{a\in A} \pi_\theta(a|s) q_i(s,a) \right), \tag{46}$$

$$A = \{a : M(s,a) > 0\}. \tag{47}$$

The value $v_{\text{mix}}(s)$ is intended as a more stable estimator of $V^{\pi_\theta}(s)$ than the single DNN prediction $v_\phi(s)$. [24] showed that using estimators of this form preserves policy improvement.

This distills to the following return values:

$$\pi_{\text{search}}^{\text{GMZ}}(a|s_1) = \mathcal{I}_{\text{reg}}(\pi_\theta, q_M), \quad V_{\text{search}}(s_1) = \sum_{a\in\mathcal{A}} \pi_{\text{search}}^{\text{GMZ}}(a|s_1) q_M(s_1, a), \tag{48}$$

$$a \sim \pi_{\text{search}}^{\text{classical}}(s_1), \quad \text{or} \quad a = \text{SH recommended action}. \tag{49}$$

Putting it all together Recent work from the SMC for search literature [34] uses SH only for search budget allocation at the root s_1 to great benefit. Since this does not bias the search, it maintains the same policy improvement properties of π_{search}.

However, the completed Qs, although sound for policy improvement, may result in large mass for unvisited actions. This results in potentially a very high entropy policy for action selection in the environment, which may be catastrophic in environments where one bad move can terminate the agent (falling of cliffs, or catastrophic moves in boardgames).

Putting all of this together, we opt for the following:

$$\pi_{\text{search}}(a|s_1) = \mathcal{I}(\pi_\theta, q_M), \quad V_{\text{search}}(s_1) = \sum_{a\in\mathcal{A}} \bar{\pi}(a|s_1) q_M(s_1, a), \quad a \sim \bar{\pi}(s_1). \tag{50}$$

$$\bar{\pi}(a|s_1) \propto \begin{cases} \pi_{\text{search}}(a|s_1) & \text{if:} \quad M(s_1,a) > 0 \\ 0 & \text{if:} \quad M(s_1,a) = 0 \end{cases} \tag{51}$$

The policy $\bar{\pi}$ has mass only on visited actions, and otherwise is proportional to π_{search}. As a result, the root value $V_{\text{search}}(s_1)$ only incorporates q_M of visited actions (e.g. no completed Q values participate in the value computation). Similarly, the acting policy is not over-entropic and only incorporates information from visited actions (e.g. again, no completedQ values participate in the value computation). $\bar{\pi}_{\text{search}}$ and can be thought of as a more principled way to achieve the sampling used by GumbelMCTS $a \sim \pi^{\text{classical}}_{\text{search}}(s_1)$.

Using V_{search} to generate value targets [29] has been shown to be sound by [56], despite V_{search} not being the value of the policy π_θ.

H.3 The Virtual Losses and Virtual Means Heuristics

We use two popular variations of *virtual visits*: Virtual Means (VM) [22] and Virtual Losses (VL) [18, 49]. Under these heuristics, the deterministic search policy is computed as follows:

$$\pi^{\text{PUCT}}_{\text{VL},i}(s) := \arg\max_{a \in A} q_i^{\text{VL}}(s_t, a) + \pi_\theta(a|s_t)\, C \frac{\sqrt{M_i(s_t) + M_i^{\text{virtual}}(s_t)}}{1 + M_i(s_t, a) + M_i^{\text{virtual}}(s_t, a)}. \tag{52}$$

$$\pi^{\text{PUCT}}_{\text{VM},i}(s) := \arg\max_{a \in A} q_i(s_t, a) + \pi_\theta(a|s_t)\, C \frac{\sqrt{M_i(s_t) + M_i^{\text{virtual}}(s_t)}}{1 + M_i(s_t, a) + M_i^{\text{virtual}}(s_t, a)}. \tag{53}$$

This is the PUCT formula [23] but including the virtual visits added by unevaluated particles / processes. q^{LV} is q_i except each virtual visit is treated as a *loss*, i.e. a return of -1. We keep the implementation of VM and VL consistent with mctx's [43] PUCT implementation: the Q values of unvisited actions are completed with the value prediction $v_\phi(s)$ and normalized after virtual losses are added. After normalization, the normalized Q-values for unvisited actions are set to 0.

H.4 Sequential Halving with P/MCTS

GumbelMCTS [24] modifies the search at the root of the tree with Sequential Halving (SH, [44]). The SH algorithm is a budget-allocation algorithm, which takes as input *number of actions to search* K and divides the search into $\log_2 K$ iterations. Each iteration is assigned a search budget $B = M/\log_2 K$. At each iteration $i \geq 1$, the number of searched actions halves $K_i = K/2^{i-1}$, and assigned equal budget: B/K_i.

When implementing SH with particles, the budget is treated as MN instead of N, and divided accordingly. To keep the parallelization across N intact, N particles are used during search at every iteration. See [34] for more detail.

SH interacts with PMCTS: on the one hand, it allows PMCTS to drive diversity by setting large $K = N$ for example. On the other hand, in later iterations, all particles are forced to search the same actions, which can reduce trajectory diversity. As a result, it's not clear whether SH should increase performance or reduce performance when coupled with PMCTS. In our experiments, we were not able to see benefit from using a parallel, particle-based variation of SH at the root of PMCTS.

H.5 Baselines

All baselines are implemented within our PMCTS framework. Virtual Means / Losses are thus also implemented as batch-parallel and synchronous. This means that all "particles" (what would be threads or processes) have to select trajectories fully sequentially in order to communicate the virtual losses added. If we attempted to select all actions simultaneously, all "particles" would pick the same action, since Virtual Means / Losses are deterministic. Their backpropagation is the same as used by PMCTS, but without the PMCTS-specific features such as ESS, de-duplication, etc. As a result, Virtual Means / Losses only parallelize the expansion well, which is not the best case for these algorithms. In more standard implementations of Virtual Means / Losses, selection and backup are done asynchronously with atomics or locks to deal with the situation where multiple threads are at the same node at the same time. Even in these implementations however, in the worst case, when *all* threads are at the same node at the same time, the selection is effectively sequential.

GumbelMCTS and PUCT MCTS are implemented in our setup, to keep comparison as direct as possible, based on the official implementation [43]. We have tested our implementation against [43] and match it almost perfectly, with remaining differences attributed to floating point inaccuracies.

Root-parallelization with GumbelMCTS We conduct independent search on N copies of the state with different random keys and then aggregate the results. In continuous action environments the "action space" at the root is shared by sampling K actions from π_θ once and treating this as a discrete action space [31]. From that point on, every search tree is unique. Once search has concluded, we aggregate across search trees by taking either the mean of the Q values (in continuous environments) or the mean of the policy (in board games): $\bar{q}_M(s,a) = \frac{1}{N}\sum_{i=1}^{N} q_M^i(s,a)$ or $\bar{\pi} = \frac{1}{N}\sum_{i=1}^{N} \pi_M^i$.

I Experimental Details

I.1 Environments and Scaling Experiments Across Domains

We evaluate PMCTS empirically across four domains and 10 environments: four two-player zero-sum board games (chess, Gardner chess, 19x19 Go and 9x9 Go) implemented by PGX [39]; Two standard discrete-control single-agent problems (Snake and RubiksCube) implemented by Jumanji [40]; Three classical continuous control tasks (Ant, Humanoid and HalfCheetah) implemented by Brax [41]. One LLM-scaling inference in the environment Sokoban [40] described in more detail in Appendix I.8.

We have DNN models (v_ϕ and π_θ pairs) for each environment: In 9x9 Go and Gardner chess we use the baseline models provided by PGX. For full chess and 19x19 Go we trained our own networks as PGX does not supply those (architecture is described in Appendix J.2). In the discrete and continuous control environments we use three models (three pairs), trained independently for different numbers of steps. We use checkpoints prior to convergence to keep the evaluation as meaningful as possible.

For each environment, agent, particle budget N and model, we run multiple evaluation episodes. In the board game environments, this amounted to each variant (agent and particle budget N) playing multiple games against each other variant, and computing Bayes Elo, which also provides a 95% confidence intervals. The games started from positions in an opening book, generation of which is explained in I.2. In the single-player environments, this amounted to each variant (environment, agent, particle budget N and model) running 256 episodes. We then compute the mean and a ± 2-SEM based $> 95\%$ confidence interval. The SEM is across both models and games (Figure 6). In the aggregated results presented in Figure 1, each individual run is first normalized between minimum possible return (0 in these environments) and maximum observed across all agents in each environments. Then the results are aggregated per agent and a similar mean and SEM-based $> 95\%$ confidence bound is computed and presented.

I.2 Opening Book

All agents in the board-game scaling experiments are deterministic. Starting every match from the initial position would make all matches identical, so we use an opening book: a set of starting states. Each environment's book contains 20,000 unique positions, generated by sampling from the policy network up to a predetermined move count, then filtering for roughly balanced positions. A position is roughly balanced when that model's value prediction lies in $[-0.3, 0.3]$. The number of moves sampled per environment is given in Table 2.

I.3 Acting in the Environment

The different agents act as follows: PMCTS selects the actions which maximizes the improved policy at the root: $\arg\max_a \pi_{search}(a|s_1)$. Virtual Means / Losses act in the standard manner of this heuristic, by taking the action with the most visitations: $\arg\max_a M(s_1, a)$. The aggregated GumbelMCTS is implemented as taking the action which maximizes the *aggregated* improved policy (Appendix H.5). See Figure 8 for the effect of action selection for each agent.

In the learning experiment, GumbelMCTS picks an action stochastically from the improved policy in the first 8 moves, and after that picks deterministically the action recommended by SH. Similarly, in the learning experiment PMCTS selects by sampling from $\pi_{search}(s)$ at the first 8 actions, and then deterministically taking the $\arg\max_a \bar{\pi}(a|s)$ (see Appendix H.2).

I.4 Evaluation Metrics

In the standard RL environments, we use the observed averaged return across multiple independent evaluation episodes. In the board game environments, we use BayesElo [42], a commonly used metric for comparing the relative strength of agents in two-player zero-sum games. Elo rating difference is related to the win-rate between agents. Bayes Elo finds the maximum likelihood ratings using MM [57] where the prior on the Elo distributions is assumed to be uniform. Elo is shift agnostic: only the relative Elo between agents matters, not the absolute score. To compute the average Elo across board games we shift a single agent to 0 in every environment (the $N = 1$ Virtual Means), shift the Elo scores of all other agents in each environment accordingly, and then take the average across environments per agent. For the policy improvement experiment in Go, we plot the win-rate against the PGX baseline agent.

We configure BayesElo with first-player advantage fixed at zero, since whenever we simulate a match between two agents, we play the the same opening from both sides. Reported intervals are BayesElo's 95% confidence intervals.

I.5 Discrete vs. Continuous Action Spaces

To extend PMCTS and MCTS to continuous actions we follow the popular approach of SampledMZ [31], which samples C actions from π_θ at each node in the search tree and treats $\{a_1, \ldots, a_C\}$ as a discrete action space with uniform logits.

I.6 Runtime Experiments

We measured the runtime for every algorithm at every particle budget (powers of 2 from 1 to 128) with a simulation budget of 128. Each measurement consisted of 3 warmup runs to remove the overhead of JIT compilation, followed by 10 trials of 10 runs each. Every run used a different input state (generated ahead of time) to reflect real use.

We plot the mean across trials, in effect the mean across all runs, and the 95% confidence intervals computed from the trials.

I.7 Policy Improvement Experiment

We use [39] implementation of a simple on-policy AlphaZero training loop for the policy improvement experiments on board games. The training hyperparameters (learning rate, batch sizes, etc.) were tuned for the baseline GumbelMCTS based AlphaZero, and were not modified and thus are likely to favor the baseline agent (purple line in Figure 3, right).

I.8 LLM inference scaling on Sokoban

In this experiment, the LLM is playing instances of the game of Sokoban in which we use text-based rendering (ASCII). Each Sokoban instance is a procedurally generated solvable 1-box puzzle. The model we used was Qwen2.5-14B-Instruct. The LLM supplies the prior policy over the actions to search at each step. For the value approximation at the leaves uniform random rollouts are used with a true model of the environment.

The system prompts used were:

You are solving a Sokoban puzzle. You control the player (P) on a grid. Your goal: push all boxes (X) onto the targets (O).

Symbols: # wall, _ floor, O target, P player X box, V box on target, S player on target.

Rules:

1. Move up, down, left, or right.
2. Push a box by moving into it (space behind must be empty).
3. You cannot pull boxes or push two at once.
4. Boxes cannot be pulled back. A box pushed into a corner or against a wall may be stuck forever — plan multiple steps ahead before pushing.

First reason about which moves are useful inside <think>...</think> tags. Then choose exactly one action inside <action>...</action> tags. Example: <think>The box is above me and the target is above the box. Pushing up will place it on the target without blocking anything.</think><action>up</action>

Then in each step, the system prompt is concatenated with this message:

Step {step_count} of {max_steps}.

{board_text}

Moves so far: {action, action, ...} (only present if move_history is non-empty).

Your move?

We do not concatenate the entire history into the LLM as that would results in very expensive runtime.

J Hyperparameters

J.1 Hyperparameter tuning

All hyperparameters except the temperature parameter η introduced in this work follow [24] as published in [43]. Search over η is presented in Figure 3, where $\eta = 1.5$ dominates all other constant temperatures. We leave the investigation of more complex temperature schedules depending on state, policy entropy, number of particles at the state, etc. to future work.

J.2 Network architectures

Different architectures are used in different environments. In the discrete-action and continuous-action single-agent environments, we use the architectures used by [38]. These are MLPs in all environments except Snake, where a CNN followed by an MLP is used. See [38] for a full specification.

In the board game experiments we use the PGX [39] baseline models for 9x9 Go and Gardner chess: small AlphaZero-style ResNets with 6 layers and 128 filters. For chess and 19x19 Go where we trained our own models, we used the same architecture as the PGX models, except for 19x19 Go we used 16 residual blocks instead of 6.

J.3 Hyperparameters

This subsection covers the hyperparameters used for the experiments. **??** gives the hyperparameters shared among experiments: These are the PUCT parameters used by Virtual Means and Virtual Losses, and the Q-normalization used by both PMCTS and Gumbel MCTS. Table 2 lists hyperparameters specific to the experiments in 9x9 Go and Gardner Chess. Table 4 details the episode length per environment used. The sampled actions refer to how many actions we sample from the continuous action space for search. Finally, Table 5 gives hyperparameters for the policy optimization experiment.

Table 1: Shared Hyperparameters for All Experiments

Category	Hyperparameter	Value
PUCT (used by Virtual Means / Losses)	c_{base}	19652
	c_{init}	1.25
Q-value Scaling for the Improved Policy	β_i	$(c_{\text{visit}} + \max_a M_i(s,a))\, c_{\text{scale}}$
	c_{visit}	50
	c_{scale}	0.1

Table 2: Hyperparameters for Board Game Experiments

Category	**Hyperparameter**	**Value**
Gumbel MCTS	Sampled Actions	16
	Gumbel Scale	0.0
PMCTS	Temperature η	1.5
Model	Model for 9x9 Go	PGX baseline v0
	Model for Gardner Chess	PGX baseline v0
Opening Book	Size per Environment	20000
	Allowed Value Range	$[-0.3, 0.3]$
	Steps for 9x9 Go	12
	Steps for 19x19 Go	12
	Steps for Gardner Chess	8
	Steps for Chess	8

Table 3: Training hyperparameters for the chess and 19x19 Go evaluation networks.

Hyperparameter	Chess	19x19 Go
Search used in self-play	GumbelMCTS	GumbelMCTS
Simulation budget M	24	32
Considered actions	8	16
Discount	0.997	0.997
Self-play steps per iteration	256	512
Self-play batch size	1024	1024
Training batch size	256	512
Learning rate	10^{-3}	10^{-4}
Momentum	-	0.9
Training iterations	250	670

Table 4: Hyperparameters for Continuous and Discrete Environments

Environment	**Type**	**Episode Length**	**Bootstrapped Actions**
Snake	Discrete (jumanji)	4000	-
RubiksCube	Multi-Discrete (jumanji)	20	-
Ant	Continuous (brax)	1000	30
HalfCheetah	Continuous (brax)	1000	30
Humanoid	Continuous (brax)	1000	30

Table 5: Hyperparameters for Policy Optimization Experiment (AlphaZero Training)

Category	**Hyperparameter**	**Value**
Network	Architecture	ResNet V2
	Number of Filters	128
	Number of Residual Blocks	6
Self-Play	Batch Size	1024
	Simulations per Step	24
	Max Number of Steps	256
	Random Plies	8
Gumbel MCTS	Sampled Actions	8
	Gumbel Scale	1.0
PMCTS	Number of Particles	4
	Temperature η	1.5
Training	Batch Size	256
	Discount	0.997
	Optimizer	Adam
	Learning Rate	0.001