SIMULATION-BASED SEARCH

DAVID SILVER AND ANDRE BARRETO

ABSTRACT

Planning is one of the oldest and most important problems in artificial intelligence. Simulation-based search algorithms, such as AlphaZero, have achieved superhuman performance in chess and Go and are used widely in real-world applications of planning. In this paper we provide a unified framework for simulation-based search. Algorithms in this framework interleave operators for policy evaluation (better estimating the value function of the current policy) and policy improvement (using the value function to form a better policy). These operators are applied to states and actions that are sampled in sequential trajectories, and that may branch recursively into other sampled trajectories. The value function and policy may also be represented by a function approximator. Our framework includes a broad family of search algorithms that includes Monte-Carlo tree search, sparse sampling, nested Monte-Carlo search, classification-based policy iteration, and AlphaZero.

MATHEMATICS SUBJECT CLASSIFICATION 2020

Primary 68T05; Secondary 68T20, 68T37, 90C39, 90C40

KEYWORDS

Planning, reinforcement learning, Markov decision processes, Monte-Carlo tree search, Monte-Carlo simulation

INTERNATIONAL CONGRESS OF MATHEMATICIANS 2022 JULY 6—14

© 2022 International Mathematical Union
Proc. Int. Cong. Math. 2022, Vol. 6, pp. 4800–4819
DOI 10.4171/ICM2022/180

1. INTRODUCTION

One of the oldest and most important problems in artificial intelligence is to select an action by looking ahead. Such planning methods are ubiquitous across several decades of artificial intelligence research, and have achieved superhuman level performance in chess [9] and Go [34], as well as contributing to real-world applications such as process control [15], robotics [24], and logistics [26].

Planning algorithms may be described by successive applications of operators that propagate information from subsequent states back to a previous state. The nature of planning is determined both by the nature of those operators, and by the order in which they are applied. A large family of planning algorithms may be understood as instances of *generalized policy iteration* [39]. In these algorithms, operators alternate between policy evaluation (better estimating the value function of the current policy) and policy improvement (using the value function to form a better policy). If operators of both types are applied repeatedly to all states, this procedure will result in the optimal value function and an optimal policy for any Markov decision process.

The precise order in which operators are applied, known as the *search control* method, has a significant impact on the efficiency of the algorithm. While many approaches to search control exist, we focus on simulation-based search. In this approach, actions and state transitions are sampled in sequential trajectories; this allows simulation-based search algorithms to look many steps ahead. Simulation-based search algorithms such as Monte-Carlo tree search [12] have been successful in large and complex planning problems such as the game of Go. We also consider recursive simulation algorithms that branch into many child simulations before backtracking to the parent. This ensures that the dependencies of an operator are accurately computed, by recursive simulation, before that operator is applied.

Many complex problems are intractable to exact solution. The state space may be too large to explicitly represent all states. In this case, the value function or policy may be represented by a function approximator such as a neural network. We show that some of today's most powerful planning algorithms, such as AlphaZero [36], can be understood as instances of generalized policy iteration using recursive simulation and function approximation.

The contribution of this paper is a unified understanding of simulation-based search algorithms. Algorithms in this framework are elucidated by three complementary mechanisms. First, equations are provided, based on the application of operators to the joint space of value functions and policies. Second, diagrams are provided, akin to backup diagrams [39], that show the states and actions used by an operator. Third, pseudocode is provided for several key algorithms.

2. RELATED WORK

Extensive literature exists on policy iteration and value iteration methods, e.g., [6,27]. However, there is little discussion of search control in this literature. The relationship of policy and value iteration to AlphaZero is discussed in [5], including an elegant exposition of their relationship to Newton's method. However, search control is not discussed in depth.

Operators that act upon a value function, such as the Bellman operator, are thoroughly analyzed within the dynamic programming literature [6,27]. Several operators that act upon a policy are introduced in [17]. Generalized operators that unify the treatment of maximizing and minimaximizing operators, among others, are discussed in [23]. The treatment of generalized policy iteration using operators that act jointly upon a value function and policy may be novel to this paper.

Tree-based search algorithms are extensively analyzed in the search literature, e.g., [10]. Simulation-based search algorithms are discussed in [7] and their relationship to reinforcement learning is discussed in [31,32]. Several search algorithms have combined elements of both depth-first search and simulation, e.g., [11,29,32,36] but there has been little prior discussion of the common principles underlying these algorithms.

3. OPERATORS

We consider a discounted Markov decision process (MDP) with a finite state space $\mathcal{S}$ and a finite action space $\mathcal{A}$ [27]. The MDP has reward and transition dynamics $R, S' \sim \varepsilon(s, a)$, where ε is a joint probability distribution over reward $R \in \mathbb{R}$ and next state $S' \in \mathcal{S}$ conditioned on current state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$. The discount factor of the MDP is $\gamma \in [0, 1)$. A policy $\pi : \mathcal{S} \to \Delta^{|\mathcal{A}|-1}$ specifies the probability of selecting each action $a \in \mathcal{S}$ in every state $s \in \mathcal{S}$; here $\Delta^{|\mathcal{A}|-1}$ is the probability simplex of dimension $|\mathcal{A}| - 1$. We will use $\mathbb{\Pi}$ to denote a complete metric space composed of all possible policies.

Let $\mathbb{Q}$ be a complete metric space whose elements are action–value functions $q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that map a state $s \in \mathcal{S}$ and an action $a \in \mathcal{A}$ onto a scalar value. We will henceforth simply refer to action-value functions as value functions. The true value function[1] of a policy π, denoted by $q_\pi \in \mathbb{Q}$, is defined as $q_\pi(s, a) = \mathbb{E}_{\pi,\varepsilon}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$, where $\mathbb{E}_{\pi,\varepsilon}[\cdot]$ denotes expectation over the Markov process $S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, \ldots$ where $R_{t+1}, S_{t+1} \sim \varepsilon(S_t, A_t)$ and $A_t \sim \pi(S_t)$ for $t = 1, 2, \ldots$ The optimal value function is defined as $q^*(s, a) = \max_{\pi \in \mathbb{\Pi}} q_\pi(s, a)$ for all $(s, a) \in \mathcal{S} \times \mathcal{A}$; it is well known that such a function always exists and is unique [27]. An optimal policy π^* is any policy that achieves the maximum value for all states and actions, that is, any policy whose value function is q^*.

We consider *operators* $\boldsymbol{o} : \mathbb{Q} \times \mathbb{\Pi} \to \mathbb{Q} \times \mathbb{\Pi}$ that transform functions and policies into other functions and policies. When the same operator is applied multiple times, we use the shorthand $\boldsymbol{o}^n(q, \pi) = (\boldsymbol{o}\boldsymbol{o} \ldots \boldsymbol{o})(q, \pi)$. When composing different operators, we use $(\prod_{i=1}^{n} \boldsymbol{o}_n)(q, \pi)$ to denote the sequence of operators $\boldsymbol{o}_1 \boldsymbol{o}_2 \ldots \boldsymbol{o}_n$ applied to (q, π) from right to left, $(\prod_{i=1}^{n} \boldsymbol{o}_n)(q, \pi) = (\boldsymbol{o}_1 \boldsymbol{o}_2 \ldots \boldsymbol{o}_n)(q, \pi) = \boldsymbol{o}_1(\ldots(\boldsymbol{o}_n(q, \pi)))$.

1 We will refer to any $q \in \mathbb{Q}$ as a value function, and use the term "true" value function to denote the specific value function corresponding to an expected return.

We begin with two primitive operators: *evaluation* operators that update the function q, but leave the policy π unchanged, and *improvement* operators that update π but leave q unchanged. An evaluation operator $\boldsymbol{e}(q, \pi)$ has the property that it moves q closer to the value function of π, such that a sequence of n applications of that operator converges to the true value function of π as n grows, $\lim_{n\to\infty} \boldsymbol{e}^n(q, \pi) = (q_\pi, \pi)$. A canonical example of an evaluation operator would be the one-step Bellman expectation operator, $\boldsymbol{e_B}(q, \pi) = (Bq, \pi)$ where $(Bq)(s, a) = \mathbb{E}_{R,S'\sim\mathcal{E}(s,a),A'\sim\pi(S')}[R + \gamma q(S', A')]$.

An improvement operator $\boldsymbol{i}$ is an operator that, when applied to a policy π and its value function q_π, produces a new policy π' whose value function is at least as large, for all state-action pairs, as that of π: $\boldsymbol{i}(q_\pi, \pi) = (q_\pi, \pi')$ such that $q_{\pi'} \geq q_\pi$ with equality if and only if $q_\pi = q^*$. A canonical example of an improvement operator would be the greedy operator, $\boldsymbol{i}^*(q_\pi, \pi)$, which produces a new policy defined as $\pi'(s) \in \operatorname{argmax}_{a\in\mathcal{A}} q_\pi(s, a)$ for all $s \in \mathcal{S}$; however, many other improvement operators are possible [17].

By alternating evaluation and improvement operators, one can compute an optimal policy for an MDP. For example, the well known value iteration algorithm can be understood as successive applications of the operators $\boldsymbol{e_B}$ and $\boldsymbol{i}^*$; starting from any function q, the sequence $(\boldsymbol{e_B}\boldsymbol{i}^*)^n(q, \cdot)$ approaches (q^*, π^*) as $n \to \infty$. If instead we consider the sequence $(\boldsymbol{i}^*\boldsymbol{e}_{\boldsymbol{B}}^{\infty})^n(\cdot, \pi)$, we obtain the well known policy iteration algorithm, which converges to (q^*, π^*) in a finite number of iterations.

3.1. State and state–action operators

Interesting problems typically contain many state variables (that is, the state space $\mathcal{S}$ is high-dimensional). The size of $\mathcal{S}$ grows exponentially with the number of variables, an issue known as the curse of dimensionality. Consequently, it may be infeasible to update all state-action pairs.

However, it is not necessary to apply operators to the entire state space at once. We will use $\boldsymbol{o}[s]$ or $\boldsymbol{o}[s, a](q, \pi)$ to denote a *state* or *state–action operator* that is specific to that state or state–action. An evaluation operator $(q', \cdot) = \boldsymbol{e}(q, \cdot)$ has a corresponding state–action evaluation operator $(q'', \cdot) = \boldsymbol{e}[s, a](q, \cdot)$ where the resulting value function q'' is only modified at a single state–action (s, a) and not modified for other states and actions,

$$q''(\bar{s}, \bar{a}) = \begin{cases} q'(s, a) & \text{if } \bar{s} = s \text{ and } \bar{a} = a, \\ q(s, a) & \text{otherwise.} \end{cases}$$

Similarly, an improvement operator $\boldsymbol{i}[s]$ improves the policy only at that single state s, and does not modify the policy for other states.

We will also encounter other operators $\boldsymbol{x}[s]$ or $\boldsymbol{x}[s, a] = \boldsymbol{o}[s_1, a_1] \ldots \boldsymbol{o}[s_n, a_n]$ that are indexed by a state s, state–action (s, a), or other variables. These operators may be composed internally of multiple state–action operators and hence may modify the value function and policy at multiple states.

State–action operators provide great flexibility on how pairs (q, π) are updated. Let $\boldsymbol{i}$ be a generic improvement operator and let $\boldsymbol{e}$ be a generic evaluation operator. Given any sequence of states and actions $(s_1, a_1), (s_2, a_2), \ldots$ that includes all state–action pairs infinitely many times, the application of $\prod_{i=1}^{n}(\boldsymbol{i}\,\boldsymbol{e})[s_i, a_i](q, \pi)$ will approach (q^*, π^*) as $n \to \infty$. The order in which state–action pairs show up in the sequence can have a significant impact on the convergence rate. This flexible way of updating (q, π) is called *generalized policy iteration* [39].

3.2. Sample-based evaluation

The curse of dimensionality also means that computing an expectation over all successor states, such as the one appearing in the evaluation operator $\boldsymbol{e_B}$, may be infeasible. *Sample-based* evaluation operators address this issue by estimating the true value function using samples from the distribution underlying the expectation. This is achieved by decomposing evaluation into two steps: constructing a return from a sampled trajectory, and updating the value function towards the return. These two steps may be represented by a *return operator* and a *value update operator,* respectively. These operators are applied to triples (q, π, g) rather than pairs (q, π). The additional argument to the operator is a scalar, $g \in \mathbb{R}$, that is used to maintain a sample of a return, such as the total discounted return $R + \gamma R' + \gamma^2 R'' + \cdots$, that follows a state–action pair (s, a) when executing policy π.

A return operator $\boldsymbol{r}$ only modifies g, leaving q and π unaltered. The *Monte-Carlo return operator* is defined as $\boldsymbol{r}_1[r, s'](q, \pi, g) = (q, \pi, r + \gamma g)$, where r is the reward and s' is a state. When applied repeatedly to the transitions of a trajectory $S_t, A_t, R_{t+1}, \ldots, S_T$ it produces the total discounted return, $\boldsymbol{r}_1[R_{t+1}, S_{t+1}] \ldots \boldsymbol{r}_1[R_T, S_T](q, \pi, 0) = (q, \pi, \sum_{j=1}^{T-t} \gamma^{j-1} R_{t+j})$, recalling that a sequence of operators is applied from right to left, corresponding to the application of operator $\boldsymbol{r}_1$ over the sequence in reverse order.[2] More generally, we define the *λ-return operator* $\boldsymbol{r}_\lambda$ as

$$\boldsymbol{r}_\lambda[r, s'](q, \pi, g) = \big(q, \pi, r + \gamma\big(\lambda g + (1-\lambda)v(s')\big)\big) \tag{1}$$

where $v(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q(s, a)$ is the value of state s.

When applied to a sampled trajectory $S_1, A_1, R_1, \ldots, S_T$ the λ-return operator computes a geometrically weighted mixture, $(1-\lambda)\sum_{l=1}^{T-t-1} \lambda^{l-1} G_{t:t+l} + \lambda^{T-l-1} G_{t:T}$, of l-step returns, $G_{t:t+l} = \sum_{j=t}^{l-1} \gamma^{j-t} R_{j+1} + \gamma^{l-1} v(S_{t+l})$ [38], which as a special case includes the Monte-Carlo return operator $\boldsymbol{r}_1$ when $\lambda = 1$.

Equipped with the concept of return operators, we can now introduce value update operators. These operators update the value function to approximate the return g. A canonical example of a value update operator adjusts the value function q by a fixed[3] step-size $\alpha \in (0, 1]$ in the direction of the return g, $\boldsymbol{e}_q[s, a](q, \pi, g) = (q', \pi, g)$ where $q'(s, a) = q(s, a) + \alpha(g - q(s, a))$. By composing the update with the λ-return operator, $\boldsymbol{e}_q[s, a]\boldsymbol{r}_\lambda[r, s'](q, \pi, g)$, we recover the well known temporal difference update TD(λ).

2 One could also conceive other operators, such as eligibility traces [39], that are applied to the sequence in forward order.

3 In practical algorithms, step-sizes may vary or adapt over time.

We can make any improvement operator $\boldsymbol{i}$ "compatible" with sample-based evaluation operators by simply defining $\boldsymbol{i}'(q, \pi, g) = (q, \pi', g)$, where $(\cdot, \pi') = \boldsymbol{i}(\cdot, \pi)$. In what follows we will focus exclusively on operators that operate over triples (q, π, g). We will henceforth use $\boldsymbol{i}$ as a generic improvement operator defined over (q, π, g); this may be instantiated by any improvement operator, such as the greedy improvement operator $\boldsymbol{i}^*$, or a policy gradient operator [40]. We will illustrate algorithms using value update operator $\boldsymbol{e}_q$ and a λ-return operator $\boldsymbol{r}_\lambda$, although these could in practice be replaced by other value update and return operators.

4. SEARCH CONTROL

We now turn our attention to the order in which states and actions are visited, so that operators may be applied in an efficient sequence.

Because the state space is typically very large, it is often infeasible to compute the optimal value function and optimal policy for all states. Instead, planning methods often focus upon the computation of the optimal value $q^*(s, \cdot)$ and an optimal policy $\pi^*(\cdot|s)$ for a specific state s. To solve this problem, it is sufficient to solve a local MDP $\mathcal{M}[s]$ consisting of a subset of states in the original MDP M that are reachable from state s with nonzero probability. The local MDP otherwise has the same action space, reward, and transition dynamics as the original MDP M.

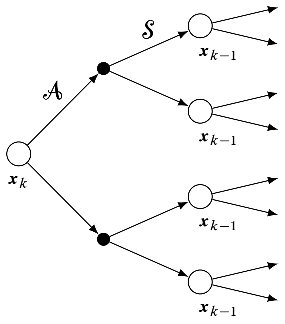
Ideally, we would like to have algorithms that are guaranteed to solve the local planning problem. We will define a *sound* planning operator $\boldsymbol{x}[s]$ to be one that converges with repeated application to the optimal value function and an optimal policy (q^*, π^*) for the local MDP $\mathcal{M}[s]$,

$$\lim_{n\to\infty} \boldsymbol{x}[s]^n(q, \pi, \cdot) = (q^*, \pi^*, \cdot), \quad \text{for any } q \in \mathbb{Q}, \pi \in \Pi. \tag{2}$$

Since generalized policy iteration finds the solution to general MDPs, it is also a sound algorithm for local planning, so long as all reachable states and actions are visited infinitely often. Note that the set of reachable states may be dramatically smaller than in the complete problem. For example, only a tiny fraction of possible positions in chess are reachable from a given endgame position; solving that endgame may be considerably simpler than solving the entire game. However, the number of reachable states may nevertheless still be large, so the order in which those states are visited remains of crucial importance.

4.1. Backup diagrams for search control

A *search control* strategy determines the order in which operators are applied to states and state–actions. It may be illustrated by a *backup diagram* [39] that shows the states and actions used by a search operator $\boldsymbol{x}[s]$. In these backups diagrams, large white circles represent states and small black circles represent state–actions. Arrows indicate transitions from state to state–action and from state–action to state. If arrows are labeled by the action space $\mathcal{A}$ or by the state space $\mathcal{S}$ then this denotes corresponding transitions for all actions $a \in \mathcal{A}$ or to all states that may follow a state–action pair. If an arrow is labeled by an environment ε or policy π this denotes that the successor state or action is sampled from the corresponding environment or policy (these labels may be omitted where clear from context). At most two transitions will be shown from each state or state–action. The leftmost circle indicates the root state s to which the search operator $\boldsymbol{x}[s]$ is applied (or sometimes the root state-action s, a to which the search operator $\boldsymbol{x}[s, a]$ is applied). Some search operators $\boldsymbol{x}_k[s]$ may be indexed by a level k, and are defined recursively in terms of lower-level operators $\boldsymbol{x}_{k-1}[s']$; in this case the states in the backup diagram associated with s and s' may be labeled by the corresponding operators. For example, a depth-first search operator could be represented by the following backup diagram:



4.2. Simulation

Simulation is a search control strategy in which trajectories are generated by sampling actions from the policy and sampling next states from the environment, as represented by the following search control diagram:

$$\begin{tikzcd} \underset{S_t}{\bigcirc} \arrow[r, "\pi"] & \underset{A_t}{\bullet} \arrow[r, "\varepsilon"] & \underset{S_{t+1}}{\bigcirc} \arrow[r, "\pi"] & \underset{A_{t+1}}{\bullet} \arrow[r, "\varepsilon"] & \cdots \arrow[r] & \underset{S_T}{\blacksquare} \end{tikzcd} \tag{3}$$

Simulation ensures that the most likely future outcomes under the current policy are explored most frequently, and may provide an effective mechanism for estimating future value, even when the state space is prohibitively large for full-width tree search. A *simulation-based search* applies evaluation and improvement operators to the sequence of states and actions encountered during simulation.

To simplify the definition of the operators, we will assume that all policies eventually reach a terminal state. The operator $\boldsymbol{x}[s]$ defined below applies evaluation and improvement operators to the sequence following state s until termination,

$$\boldsymbol{x}[s](q, \pi, g) = \begin{cases} (q, \pi, 0) & \text{if } s \text{ is terminal,} \\ \boldsymbol{i}[s]\, \boldsymbol{e}_q[s, A]\, \boldsymbol{r}_\lambda[R, S']\, \boldsymbol{x}[S'](q, \pi, g) & \text{otherwise,} \end{cases}$$
$$\text{with } A \sim \pi(s) \text{ and } R, S' \sim \varepsilon(s, A). \tag{4}$$

All the operators introduced from this point on will be based on the assumption that every policy eventually terminates. When this is not the case, one can easily modify the simulation-based operators to ensure that they terminate after T steps.

Note that, unlike the evaluation, improvement and return operators previously defined, the simulation operator $\boldsymbol{x}$ potentially modifies all of its arguments (q, π, g). This operator is typically iterated over n simulations, $\boldsymbol{x}^n[s]$, to compute the value and policy at a root state $s \in \mathcal{S}$. Pseudocode for simulation-based search with sample operators is given in Algorithm 1. The function called Improve() may invoke any suitable improvement operator $\boldsymbol{i}$.

Algorithm 1 Simulation-Based Search

```
procedure Sim($\varepsilon, q, \pi, s$)
    if $s$ is terminal then return 0
    $A \sim \pi(s)$
    $R, S' \leftarrow \varepsilon(s, A)$
    $v(S') \leftarrow \sum_{a \in \mathcal{A}} \pi(a|S')q(S', a)$
    $G \leftarrow R + \gamma(\lambda\, \textsc{Sim}(\varepsilon, q, \pi, S') + (1 - \lambda)v(S'))$
    $q(s, A) \leftarrow q(s, A) + \alpha(G - q(s, A))$
    $\pi(s) \leftarrow \textsc{Improve}(\pi(s), q(s, \cdot), G)$
    return G
end procedure
```

Algorithm 2 Recursive Simulation-Based Search

```
procedure RSim($\varepsilon, q, \pi, s, k$)
    if $s$ is terminal or $k = 0$ then return 0
    for $i = 1$ to $n$ do
        RSim($\varepsilon, q, \pi, s, k - 1$)
    end for
    $A \sim \pi(s)$                                              ▷ $\pi$ may have changed
    $R, S' \leftarrow \varepsilon(s, A)$
    $v(S') \leftarrow \sum_{a \in \mathcal{A}} \pi(a|S')q(S', a)$
    $G \leftarrow R + \gamma(\lambda\, \text{R}\textsc{Sim}(\varepsilon, q, \pi, S', k) + (1 - \lambda)v(S'))$
    $q(s, A) \leftarrow q(s, A) + \alpha(G - q(s, A))$
    $\pi(s) \leftarrow \textsc{Improve}(\pi(s), q(s, \cdot), G)$
    return G
end procedure
```

One may also compute the return from a simulation without any value update or policy improvement. These simple simulations are known as *rollouts*,

$$\boldsymbol{z}[s, a](q, \pi, g) = \begin{cases} (q, \pi, 0) & \text{if } s \text{ is terminal,} \\ \boldsymbol{r}_\lambda[R, S']\, \boldsymbol{z}[S', A'](q, \pi, g) & \text{otherwise,} \end{cases}$$
$$\text{with } R, S' \sim \varepsilon(s, a) \text{ and } A' \sim \pi(S'). \tag{5}$$

Exploration with soft improvement operators. For a simulation-based search to be sound, the simulation policy π must continue to visit all states and actions infinitely often as we apply the operator multiple times. That is, if $(\cdot, \pi', \cdot) = \boldsymbol{x}^n(\cdot, \pi, \cdot)$, we want π' to select all actions with nonzero probability. This does not necessarily follow for all $\boldsymbol{x}$. For example, if we plug in the greedy improvement operator $\boldsymbol{i}^*$ in (5), the resulting π' is a deterministic policy—that is, $\pi(a|s) = 1$ for a specific $a \in \mathcal{A}$. Even when π' is not deterministic, one may want to sample actions from a distribution that ensures an appropriate level of *exploration* [39].

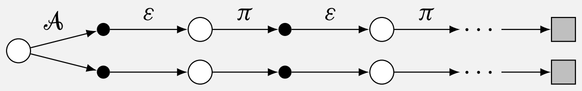
Exploration may be accomplished by using a *soft improvement operator* that yields a policy that selects all actions with nonzero probability. If the soft improvement operator approaches the greedy operator as the number of applications tends to infinity, $\lim_{n\to\infty} \boldsymbol{i}^n = \boldsymbol{i}^*$, then under mild conditions convergence is assured (c.f. (2)). This type of exploration is referred to as "greedy in the limit of infinite exploration" (GLIE) [37].

As an example, the ϵ-greedy operator $\boldsymbol{i} = \boldsymbol{\epsilon}\boldsymbol{i}^*$ is a soft improvement operator that introduces randomness via a noisy operator $\boldsymbol{\epsilon}(q, \pi, g) = (q, \epsilon\pi_{\text{rand}} + (1-\epsilon)\pi, g)$, where π_{rand} is any policy that selects all actions with nonzero probability. A common choice is to have π_{rand} select actions uniformly at random, $\pi_{\text{rand}}(\cdot|s) = 1/|\mathcal{A}|$. Note that, if we think of ϵ as a parameter of $\boldsymbol{\epsilon}(q, \pi, g)$ that is decreased at each application of $\boldsymbol{\epsilon}$, we obtain a GLIE operator. Alternatively, an upper-confidence rule $\boldsymbol{i}^{\text{UCB}}$ may be used to encourage exploration of uncertain values, by acting greedily with respect to an upper confidence bound on the value function, $\text{argmax}_{a\in\mathcal{A}}\, q(s, a) + u(s, a)$, where $u(s, a)$ represents value uncertainty [3].

In what follows we will define some operators using a generic improvement operator $\boldsymbol{i}$; unless noted otherwise, the reader should think of $\boldsymbol{i}$ as some instantiation of a GLIE soft improvement operator.

Example 4.1: All-Action Monte-Carlo Search

Historically, the earliest forms of simulation-based search [8,41], used for example to achieve superhuman performance in Scrabble [30], were based upon rollouts $\boldsymbol{z}[s, a]$ that start immediately after a single action a from state s. The main idea is to estimate the action–value of every action $a \in \mathcal{A}$ from the root state by the outcome of simulations starting from that action. Finally, the action with maximum value is executed. We refer to this search algorithm as *all-action Monte-Carlo search*, which can be described by the following search control diagram,



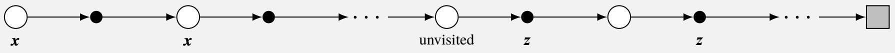
In this case a single improvement operator $\boldsymbol{i}^*$ is applied to the root state based on the values[a] estimated through the Monte-Carlo simulations:

$$\boldsymbol{x}[s](q,\pi,g) = \boldsymbol{i}^*[s] \prod_{a\in\mathcal{A}} \boldsymbol{e}_q[s,a]\boldsymbol{z}[s,a](q,\pi,g).$$

A In practice, Monte-Carlo algorithms often update the value function using a step-size $\alpha = 1/visits(s,a)$. In this case the update $\boldsymbol{e}_q$ incrementally updates the value $q(s,a)$ to the mean return following state-action s, a.

Example 4.2: Monte-Carlo Tree Search

A simulation-based search algorithm using sample operators at each state and action is known as *Monte-Carlo tree search* (MCTS) [12], as used in the first master-level 9×9 [13,16] and 19×19 [34] Go programs. Each simulation of MCTS consists of two stages: a first stage in which sample-based evaluation and improvement operators are applied, and a rollout that samples the remainder of the trajectory. The first stage of simulation typically finishes upon reaching a previously unvisited state, while the rollout finishes upon reaching a terminal state, as shown below,



$$\boldsymbol{x}[s](q,\pi,g) = \begin{cases} \boldsymbol{i}[s]\,\boldsymbol{e}_q[s,A]\,\boldsymbol{z}[s,A](q,\pi,g) & \text{if } s \text{ is unvisited,} \\ \boldsymbol{i}[s]\,\boldsymbol{e}_q[s,A]\,\boldsymbol{r}_\lambda[R,S']\,\boldsymbol{x}[S'](q,\pi,g) & \text{otherwise,} \end{cases}$$

$$\text{with } A \sim \pi(s) \text{ and } R, S' \sim \varepsilon(s,A). \qquad (6)$$

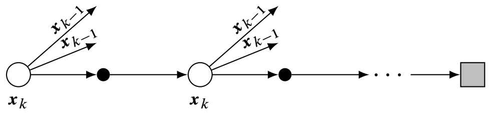
This leads to a gradual expansion of the frontier of visited states as each subsequent simulation goes one step further. When the operator $\boldsymbol{i}$ used in (6) is a GLIE soft improvement operator, MCTS is sound under mild assumptions (cf. Eq. (2)). An upper confidence bound around an estimate of q_π is often used to guide exploration [21]. As in value iteration, policies are not explicitly represented.
MCTS typically uses Monte-Carlo returns, $\boldsymbol{z}$, with $\lambda = 1$. However, a variant of MCTS that uses instead TD(λ) returns is sometimes known as *temporal-difference search* [33]

4.3. Recursive simulation

Classical search methods traverse a search tree by branching from a parent state to a child state, performing a search from the child, and then backtracking to the parent. Branching and backtracking in this manner may be advantageous because it ensures that child values are accurate before applying any operation to the parent. If the value and policy of each child are optimal, only a single operation needs to be applied to the parent to ensure optimality.

By contrast, simulation breaks the curse of dimensionality by sampling trajectories, allowing search to look deeply ahead even in large state spaces. However, it may need to visit each state multiple times.

We propose here a marriage of these two search control strategies by allowing parent simulations to branch recursively into child simulations. This produces a *recursive simulation-based search*. Each level k simulation samples a sequence of states and actions; multiple level $k-1$ simulations are invoked from each state (or state–action) of the sequence before sampling the next action. This is illustrated in the search diagram below, where each arrow labeled $\boldsymbol{x}_{k-1}$ represents a lower level simulation starting from that state, corresponding to the recursive application of the same search diagram with $k-1$,



(7)

Applying improvement and evaluation operators to the states and actions of the search results in the following search algorithm:

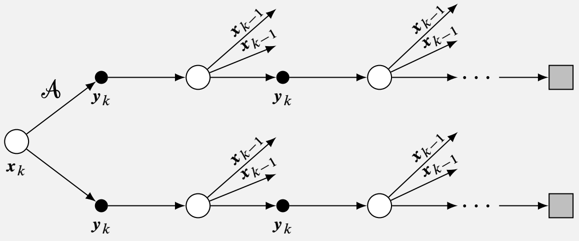
$$\boldsymbol{x}_k[s](q,\pi,g) = \begin{cases} (q,\pi,0) & \text{if } s \text{ is terminal or } k=0, \\ \boldsymbol{i}[s]\,\boldsymbol{e}_q[s,A]\,\boldsymbol{r}_\lambda[R,S']\,\boldsymbol{x}_k[S']\,\underbrace{\boldsymbol{x}^n_{k-1}[s](q,\pi,g)}_{(\cdot,\pi',\cdot)} & \text{otherwise,} \end{cases} \tag{8}$$

where $(\cdot,\pi',\cdot) = \boldsymbol{x}^n_{k-1}[s](q,\pi,g)$, $A \sim \pi'(s)$ and $R, S' \sim \varepsilon(s,A)$.

One can think of the operator above as a simulation, akin to Eq. (5), in which the action A to be taken is sampled from a policy π' resulting from the application of the operator itself. The overall algorithm, shown in Algorithm 2, is similar to Algorithm 1 with the addition of a recursive call that corresponds to the operator $\boldsymbol{x}^n_{k-1}$ in Eq. (8) (see for example *nested Monte-Carlo tree search* [4]).

Example 4.3: Nested Monte-Carlo Search

Recursive simulation may be used with an all-action Monte-Carlo search. This gives rise to the more powerful algorithm of *nested Monte-Carlo search* [11, 42]. In this algorithm, rollouts are nested within rollouts. At level $k+1$, all possible actions are enumerated. Each action $a \in \mathcal{A}$ is evaluated by the average outcome (i.e., Monte-Carlo evaluation $\boldsymbol{e}_q\boldsymbol{z}$) of level k simulations that start from action a. The policy at the root state of the simulation selects the action with maximum value (i.e., greedy improvement $\boldsymbol{i}^*$). An instance of this algorithm is illustrated by the following search control diagram and equations:



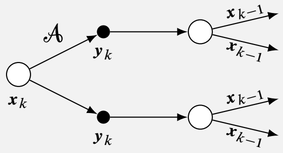
$$x_k[s](q,\pi,g) = i^*[s] \prod_{a\in\mathcal{A}} e_q[s,a] y_k[s,a](q,\pi,g),$$

$$y_k[s,a](q,\pi,g) = \begin{cases} (q,\pi,0) \text{ if } s \text{ is terminal or } k=0, \\ r_\lambda[R,S'] y_k[S',A'] x^n_{k-1}[S'](q,\pi,g) \text{ otherwise,} \end{cases}$$

$$\text{where } R,S' \sim \varepsilon(s,a), (\cdot,\pi',\cdot) = x^n_{k-1}\big[S'\big](q,\pi,g), \text{ and } A' \sim \pi'(S').$$

Nested Monte-Carlo search achieved superhuman performance in Morpion solitaire [11] and a variant of Klondike solitaire [42]. Each additional level of recursion $x_1[s],\dots,x_4[s]$ produced stronger results (even when taking account of the additional computational cost).

If simulations are truncated after one time-step, $y_k[s,a](q,\pi,g) = r_\lambda[R,S']x^n_{k-1}[S'](q,\pi,v(S'))$, then we recover a *sparse sampling* tree search algorithm [20]



5. APPROXIMATION

Many problems are so large that they are intractable to exact methods—even when using simulation. To address this issue, we consider approximate methods that use a function approximator (such as a neural network) q_θ with parameters θ to represent a value function, or a function approximator π_η with parameters η to represent a policy. We will consider operators that aim at finding the closest representable value parameters $\theta^* = \operatorname{argmin}_\theta \ell_q(q,q_\theta)$ to target value function q according to some loss ℓ_q, such as squared error, or the closest representable policy parameters $\eta^* = \operatorname{argmin}_\eta \ell_\pi(\pi,\pi_\eta)$ to a target policy π according to some loss ℓ_π, such as the KL divergence.

We formalize these operators as modified versions of their counterparts which project their operands onto the space of representable value functions or policies. They do so by directly manipulating the parameters θ or η. In practice most approximation methods incrementally optimize parameters by gradient descent. We define a gradient-based evaluation operator that acts upon an approximate value function as

$$\begin{aligned} \boldsymbol{e}_\theta(q_\theta, \pi, g) &= (q_{\theta'}, \pi, g), \\ \text{where } \theta' &= \theta - \alpha \frac{\partial}{\partial \theta} \ell_q(g, q_\theta). \end{aligned} \tag{9}$$

Gradient descent on policy parameters may be formalized analogously by defining a generic, gradient-based improvement operator,

$$\begin{aligned} \boldsymbol{i}_\eta(q, \pi_\eta, g) &= (q, \pi_{\eta'}, g), \\ \text{where } \eta' &= \eta - \alpha \frac{\partial}{\partial \eta} \ell_\pi(\pi', \pi_\eta) \quad \text{and} \quad (\cdot, \pi', \cdot) = \boldsymbol{i}(\cdot, \pi_\eta, \cdot). \end{aligned} \tag{10}$$

In the above equation, π' is the policy resulting from applying the generic improvement operator $\boldsymbol{i}$ to policy π_η. For example, a gradient-based greedy improvement operator may be instantiated, $\boldsymbol{i}^*_\eta(q, \pi_\eta, g) = (q, \pi_{\eta'}, g)$ using a corresponding greedy improvement operator $(\cdot, \pi', \cdot) = \boldsymbol{i}^*(\cdot, \pi_\eta, \cdot)$ to provide the target policy π' for the loss function ℓ_π.

Function approximation may be combined with simulation-based search by simply replacing the regular improvement and evaluation operators with their counterparts defined above.

The combination of approximation with recursive simulation-based search (see Section 4.3) yields well-known algorithms that have been successfully applied to large and complex problems, as discussed in the examples below. Algorithm 3 illustrates a canonical algorithm using function approximation.

Algorithm 3 Approximate Recursive Simulation-Based Search

```
procedure ARSim($\varepsilon$, $\theta$, $\eta$, $s$, $k$)
    if $s$ is terminal or $k = 0$ then return 0
    for $i = 1$ to $n$ do
        ARSim($\varepsilon, \theta, \eta, s, k-1$)
    end for
    $A \sim \pi_\eta(s)$                                        ▷ $\eta$ may have changed
    $R, S' \leftarrow \varepsilon(s, A)$
    $v(S') \leftarrow \sum_{a \in \mathcal{A}} \pi(a|S')q(S', a)$
    $G \leftarrow R + \gamma(\lambda\, \text{ARSim}(\varepsilon, \theta, \eta, S', k) + (1-\lambda)v(S'))$
    $\theta \leftarrow \theta - \alpha \frac{\partial}{\partial \theta} \ell_q(G, q_\theta(s, A))$
    $\pi'(s) \leftarrow \text{Improve}(\pi_\eta, q_\theta(s, \cdot), G)$
    $\eta \leftarrow \eta - \alpha \frac{\partial}{\partial \eta} \ell_\pi(\pi'(s), \pi_\eta(s))$
    return G
end procedure
```

Algorithm 4 AlphaZero

```
procedure AlphaZero($\varepsilon, \theta, \eta, s, k$)
    if $s$ is terminal or $k = 0$ then return 0
    $\pi' \leftarrow \pi_\eta; q' \leftarrow q_\theta$
    for $i = 1$ to $n$ do
        MCTS($\varepsilon, q', \pi', s, k-1$)†
    end for
    $A \sim \pi'(s)$
    $R, S' \leftarrow \varepsilon(s, A)$
    $v(S') \leftarrow \sum_{a \in \mathcal{A}} \pi(a|S')q(S', a)$
    $G \leftarrow R + \gamma(\lambda \textsc{AlphaZero}(\varepsilon, \theta, \eta, S', k) + (1-\lambda)v(S'))$
    $\theta \leftarrow \theta - \alpha \frac{\partial}{\partial \theta} \ell_q(G, q_\theta(s, A))$
    $\eta \leftarrow \eta - \alpha \frac{\partial}{\partial \eta} \ell_\pi(\pi'(s), \pi_\eta(s))$
    return G
end procedure
```

† MCTS is an instantiation of Sim (Algorithm 1).

Example 5.1: Dyna-k

The Dyna-k algorithm [32] is an example of recursive simulation-based search with value function approximation. At every level k it uses an approximate evaluation operator $\boldsymbol{e}_\theta$ that minimizes squared error with respect to a sampled return. This return corresponds to the outcome of a level k simulation in which actions are sampled from the improved policy resulting from multiple lower level $k-1$ simulations. The search control diagram follows the same pattern as Eq. (7).

$$\boldsymbol{x}_k[s](\pi, q_\theta, g) = \begin{cases} (q, \pi, 0) & \text{if } s \text{ is terminal or } k = 0, \\ \boldsymbol{i}^*[s]\boldsymbol{e}_\theta[s, A]\boldsymbol{r}_\lambda[R, S']\boldsymbol{x}_k[S']\boldsymbol{x}^n_{k-1}[s](\pi, q_{\theta'}, g), \end{cases}$$

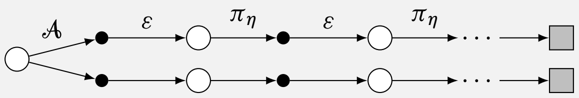
$$\text{with } \theta' = \theta, (\cdot, \pi', \cdot) = \boldsymbol{x}^n_{k-1}[s](\pi, q'_\theta, g), A \sim \pi'(s) \text{ and } R, S' \sim \varepsilon(s, A).$$

Dyna-k may also be adapted to apply function approximation to the policy (see Algorithm 3). For example, *policy gradient search* [1] utilizes a gradient-based improvement operator $\boldsymbol{i}_\eta$ based upon policy gradient algorithms [40].

In practice, the value function (or policy) is represented separately at different levels of the recursion by distinct parameters. Level $k-1$ parameters are copied from level k parameters, and are then updated based upon the level $k-1$ returns. This allows parameters to specialize to a local region of the search tree. Using multiple representations boosted performance in 9×9 Go, compared to a single representation [32].

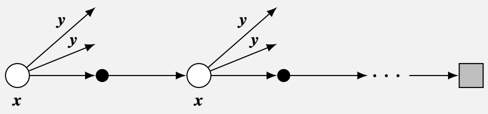
Example 5.2: Classification-Based Policy Iteration

Classification-based policy iteration (CBPI) combines all-action Monte-Carlo search with policy approximation [22]. The rollouts of the Monte-Carlo search sample actions according to π_η,



$$\boldsymbol{y}[s](q, \pi_\eta, g) = \boldsymbol{i}^*_\eta \prod_{a \in \mathcal{A}} \boldsymbol{e}_q[s, a]\boldsymbol{z}[s, a](q, \pi_\eta, g),$$

The main search consists of a second level of simulations. At every step of the main search, an all-action Monte-Carlo search is called recursively from state s, to compute an improved policy. Policy parameters η are updated by an approximate improvement operator $\boldsymbol{i}^*_\eta$ that minimizes a classification loss, $\ell_\pi(\pi', \pi_\eta)$, with respect to a greedy improvement operator $(\cdot, \pi', \cdot) = \boldsymbol{i}^*(q, \pi, g)$,



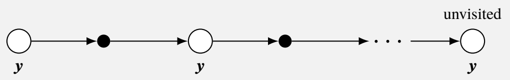
$$\boldsymbol{x}[s](q, \pi_\eta, 0) = \begin{cases} (q, \pi_\eta, 0) & \text{if } s \text{ is terminal or } k = 0, \\ \boldsymbol{x}[S']\boldsymbol{y}^n[s](q, \pi_\eta, g), \end{cases}$$

where $(\cdot, \pi_{\eta'}, \cdot) = \boldsymbol{y}^n[s](q, \pi_\eta, g)$, $A \sim \pi_{\eta'}(s)$ and $R, S' \sim \varepsilon(s, A)$.
Expert iteration applies CBPI to a Monte-Carlo tree search; it achieved state-of-the-art performance in Hex [2]. In practice, both expert iteration and CBPI are often combined with value function approximation [28].

Example 5.3: AlphaZero

The AlphaZero algorithm [35, 36] achieved superhuman performance across chess, Go and shogi. It is a two-level recursive simulation-based search that utilizes both value function and policy approximation at the second level.
At the first level, AlphaZero uses a Monte-Carlo tree search. Simulations finish upon reaching an unvisited state, without any rollout, at which point the return is initialized to the value function of this state.[a] To ensure adequate exploration, an improvement operator $\boldsymbol{i}^{\text{UCB}}$ selects the action that maximizes an upper confidence bound $u(s, a) \propto$

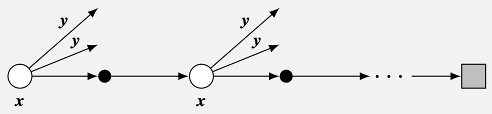
$\pi_\eta(a|s)/(visits(s,a)+1)$ that is informed by the policy π_η [34],[b]



$$\boldsymbol{y}[s](q,\pi,g) = \begin{cases} \boldsymbol{i}^{\text{UCB}}[s]\,\boldsymbol{e}_q[s,A](q,\pi,v(s)) & \text{if } s \text{ is unvisited,} \\ \boldsymbol{i}^{\text{UCB}}[s]\,\boldsymbol{e}_q[s,A]\,\boldsymbol{r}_\lambda[R,S']\,\boldsymbol{y}[S'](q,\pi,g) & \text{otherwise,} \end{cases}$$

where $A \sim \pi(s)$ and $R, S' \sim \varepsilon(s,A)$.

At the second level, AlphaZero represents its value function and policy by neural networks with parameters θ and η, respectively.[c] An approximate improvement operator $\boldsymbol{i}^\eta$ is applied to the Monte-Carlo tree search operator defined above, $\boldsymbol{y}[s]$. The operator $\boldsymbol{i}^\eta$ is based on a classification loss, similar to the one used in Example 5.2, that "projects" a policy π onto the space spanned by η. An approximate evaluation operator $\boldsymbol{e}_\theta$ is applied to the Monte-Carlo return corresponding to the outcome of the high-level simulation, in a similar manner to Algorithm 3,



$$\boldsymbol{x}[s](q_\theta,\pi_\eta,g) = \begin{cases} (q_\theta,\pi_\eta,0) & \text{if } s \text{ is terminal or } k=0, \\ \boldsymbol{e}_\theta[s,A]\boldsymbol{r}_\lambda[R,S']\boldsymbol{x}[S']\underbrace{\boldsymbol{i}_\eta[s]\boldsymbol{y}^n[s](q_\theta,\pi_\eta,g)}_{(\cdot,\pi_{\eta'},\cdot)}, & \end{cases}$$

$$\text{where } (\cdot,\pi_{\eta'},\cdot) = \boldsymbol{i}_\eta[s]\boldsymbol{y}^n[s](q_\theta,\pi_\eta,g),\ A \sim \pi_{\eta'}(s),\ S' \sim \varepsilon(s,A).$$

In reality, the operator $\boldsymbol{y}$ is called with copies of q_θ and π_η,[d] and the resulting policy is used as a target[e] for the gradient-based policy improvement π_η (see Algorithm 4).

A AlphaZero uses a state value function; for consistency, it is illustrated here using an action value function.

B Other policy improvement operators may also be used [14,18].

C In practice both neural networks share the same parameters.

D The policy copy passed into the operator $\boldsymbol{y}$ may be modified by noise, $(\cdot,\pi_{\eta'},\cdot) = \boldsymbol{i}_\eta[s]\boldsymbol{y}^n[s]\boldsymbol{\epsilon}(q_\theta,\pi_\eta,g)$; this ensures adequate exploration, even when using a small number of simulations [14].

E The target can also incorporate an improvement step based upon the outer return [19].

6. DISCUSSION

We have developed a framework for understanding simulation-based search algorithms in terms of their search control methods. We have seen that many algorithms can be described as instances of generalized policy iteration, interleaving evaluation and improvement operators to ensure convergence towards an optimal value function and policy. In large problems, approximate evaluation and improvement operators may also be introduced, so as to search for an approximately optimal value function and policy.

The formalism we proposed allowed us to describe many existing search algorithms – from a basic Monte-Carlo search to more sophisticated search algorithms such as AlphaZero – that nest multiple levels of simulation. However, many other strategies exist for search control that cannot be described in these simple terms. For example, many planning algorithms utilize *prioritization* to sort states according to an appropriate criterion [25]; the highest priority state is visited next.

We have presented several specific examples of simulation-based search algorithms that utilize simulation and recursion, including many of the most successful methods used in games such as chess and Go – the canonical challenges for planning. However, the framework presented in this paper also suggests a much broader space of search algorithms that combine elements of existing algorithms. For example, could AlphaZero [36] be improved by introducing deeper levels of recursion, as in nested Monte-Carlo search [11]? Or by utilizing function approximation inside its lower level Monte-Carlo tree search, as in Dyna-2 [32]? Could other evaluation and improvement operators be more effective [14,18,19]?

Understanding the underlying principles may also enable existing heuristic search algorithms to be replaced with sound algorithms that converge to the optimal solution under a broader range of conditions. For example, the heuristic improvement operator in AlphaZero may be replaced with a principled policy improvement operator [14]. Finally, we hope that a greater understanding of these principles may result in the development of new search algorithms that go beyond our current frontiers.

REFERENCES

[1] T. Anthony, R. Nishihara, P. Moritz, T. Salimans, and J. Schulman, Policy gradient search: Online planning and expert iteration without search trees. *CoRR* (2019), arXiv:1904.03646.

[2] T. Anthony, Z. Tian, and D. Barber, Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4–9, 2017, Long Beach, CA, USA*, edited by U. Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, pp. 5360–5370, Curran Associates Inc., 2017.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer, Finite-time analysis of the multi-armed bandit problem. *Mach. Learn.* **47** (2002), no. 2–3, 235–256.

[4] H. Baier and M. H. M. Winands, Nested Monte-Carlo tree search for online planning in large MDPs. In *20th European Conference on Artificial Intelligence* 242, edited by L. D. Raedt, C. Bessiere, D. Dubois, P. Doherty, P. Frasconi, F. Heintz, and P. J. F. Lucas, pp. 109–114, IOS Press, 2012.

[5] D. Bertsekas, Lessons from AlphaZero for optimal, model predictive, and adaptive control. 2021, arXiv:2108.10315.

[6] D. P. Bertsekas, *Dynamic Programming and Optimal Control, volume I.* 3rd edn., Athena Scientific, Belmont, MA, USA, 2005.

[7] C. Browne, E. J. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. P. Liebana, S. Samothrakis, and S. Colton, A survey of Monte Carlo tree search methods. *IEEE Trans. Computat. Intell. AI Games* **4** (2012), no. 1, 1–43.

[8] B. Bruegmann, Monte-Carlo Go, 1993, http://www.cgl.ucsf.edu/go/Programs/Gobble.html.

[9] M. Campbell, A. Hoane, and F. Hsu, Deep Blue. *Artificial Intelligence* **134** (2002), 57–83.

[10] M. Campbell and T. A. Marsland, A comparison of minimax tree search algorithms. *Artificial Intelligence* **20** (1983), 347–367.

[11] T. Cazenave, Nested Monte-Carlo search. In *21st International Joint Conference on Artificial Intelligence*, edited by C. Boutilier, pp. 456–461, International Joint Conferences on Artificial Intelligence, 2009.

[12] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search. In *5th International Conference on Computer and Games*, pp. 72–83, Springer-Verlag, 2006.

[13] R. Coulom, Computing Elo ratings of move patterns in the game of Go. In *Computer Games Workshop*, pp. 198–208, Universiteit Maastricht, 2007.

[14] I. Danihelka, A. Guez, J. Schrittwieser, and D. Silver, Policy improvement by planning with Gumbel. In *International Conference on Learning Representations*, 2022.

[15] C. E. Garcia, D. M. Prett, and M. Morari, Model predictive control: Theory and practice—A survey. *Automatica* **25** (1989), no. 3, 335–348.

[16] S. Gelly, Y. Wang, R. Munos, and O. Teytaud, Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.

[17] D. Ghosh, M. C. Machado, and N. L. Roux, An operator view of policy gradient methods. In *Advances in Neural Information Processing Systems 33*, edited by H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, pp. 3397–3406, Curran Associates Inc., 2020.

[18] J. Grill, F. Altché, Y. Tang, T. Hubert, M. Valko, I. Antonoglou, and R. Munos, Monte-Carlo tree search as regularized policy optimization. In *37th International Conference on Machine Learning 119*, pp. 3769–3778, Association of Computing Machinery, 2020.

[19] M. Hessel, I. Danihelka, F. Viola, A. Guez, S. Schmitt, L. Sifre, T. Weber, D. Silver, and H. van Hasselt, Muesli: Combining improvements in policy optimization. In *38th International Conference on Machine Learning 139*, edited by M. Meila and T. Zhang, pp. 4214–4226, Association of Computing Machinery, 2021.

[20] M. J. Kearns, Y. Mansour, and A. Y. Ng, A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *Mach. Learn.* **49** (2002), no. 2–3, 193–208.

[21] L. Kocsis and C. Szepesvari, Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning*, pp. 282–293, Springer-Verlag, 2006.

[22] M. G. Lagoudakis and R. Parr, Reinforcement learning as classification: Leveraging modern classifiers. In *20th International Conference on Machine Learning*, edited by T. Fawcett and N. Mishra, pp. 424–431, AAAI Press, 2003.

[23] M. L. Littman and C. Szepesvári, A generalized reinforcement-learning model: Convergence and applications. In *13th International Conference on Machine Learning*, edited by L. Saitta, pp. 310–318, Morgan Kaufmann, 1996.

[24] T. Lozano-Pérez, Spatial planning: A configuration space approach. In *Autonomous Robot Vehicles*, edited by I. J. Cox and G. T. Wilfong, pp. 259–271, Springer, 1990.

[25] A. W. Moore and C. G. Atkeson, Prioritized sweeping: Reinforcement learning with less data and less time. *Mach. Learn.* **13** (1993), 103–130.

[26] W. B. Powell, *Approximate dynamic programming: solving the curses of dimensionality*. 2nd edn., Wiley Ser. Probab. Stat., Wiley, Hoboken, NJ, USA, 2011.

[27] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. Wiley Ser. Probab. Stat., Wiley, 1994.

[28] B. Scherrer, M. Ghavamzadeh, V. Gabillon, B. Lesner, and M. Geist, Approximate modified policy iteration and its application to the game of tetris. *J. Mach. Learn. Res.* **16** (2015), no. 49, 1629–1676.

[29] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al., Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature* **588** (2020), no. 7839, 604–609.

[30] B. Sheppard, World-championship-caliber Scrabble. *Artificial Intelligence* **134** (2002), no. 1–2, 241–275.

[31] D. Silver, *Reinforcement Learning and Simulation-Based Search in Computer Go*. PhD thesis, University of Alberta, 2009.

[32] D. Silver, R. S. Sutton, and M. Müller, Sample-based learning and search with permanent and transient memories. In *25th International Conference on Machine Learning*, pp. 968–975, Association of Computing Machinery, 2008.

[33] D. Silver, R. S. Sutton, and M. Müller, Temporal-difference search in computer Go. *Mach. Learn.* **87** (2012), no. 2, 183–219.

[34] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. P. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, Mastering the game of Go with deep neural networks and tree search. *Nature* **529** (2016), no. 7587, 484–489.

[35] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al., Mastering the game of Go without human knowledge. *Nature* **550** (2017), no. 7676, 354–359.

[36] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al., A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* **362** (2018), no. 6419, 1140–1144.

[37] S. P. Singh, T. S. Jaakkola, M. L. Littman, and C. Szepesvári, Convergence results for single-step on-policy reinforcement-learning algorithms. *Mach. Learn.* **38** (2000), no. 3, 287–308.

[38] R. Sutton, Learning to predict by the method of temporal differences. *Mach. Learn.* **3** (1988), no. 9, 9–44.

[39] R. Sutton and A. Barto, *Reinforcement learning: an introduction*. MIT Press, 1998.

[40] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, edited by S. A. Solla, T. K. Leen, and K. Müller, pp. 1057–1063, The MIT Press, 1999.

[41] G. Tesauro and G. Galperin, On-line policy improvement using Monte-Carlo search. In *Advances in Neural Information Processing 9*, pp. 1068–1074, Curran Associates Inc., 1996.

[42] X. Yan, P. Diaconis, P. Rusmevichientong, and B. Roy, Solitaire: Man versus machine. In *Advances in Neural Information Processing Systems 17*, edited by L. Saul, Y. Weiss, and L. Bottou, MIT Press, 2004.

DAVID SILVER

DeepMind, London, UK, and University College London, London, UK, davidsilver@google.com

ANDRE BARRETO

DeepMind, London, UK, andrebarreto@google.com