Policy-Guided Heuristic Search with Guarantees

Laurent Orseau,[1] **Levi H. S. Lelis**[2]
[1]DeepMind, UK
[2]Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada
lorseau@google.com, levi.lelis@ualberta.ca

Abstract

The use of a policy and a heuristic function for guiding search can be quite effective in adversarial problems, as demonstrated by AlphaGo and its successors, which are based on the PUCT search algorithm. While PUCT can also be used to solve single-agent deterministic problems, it lacks guarantees on its search effort and it can be computationally inefficient in practice. Combining the A* algorithm with a learned heuristic function tends to work better in these domains, but A* and its variants do not use a policy. Moreover, the purpose of using A* is to find solutions of minimum cost, while we seek instead to minimize the *search loss* (*e.g.*, the number of search steps). LevinTS is guided by a policy and provides guarantees on the number of search steps that relate to the quality of the policy, but it does not make use of a heuristic function. In this work we introduce Policy-guided Heuristic Search (PHS), a novel search algorithm that uses both a heuristic function and a policy and has theoretical guarantees on the search loss that relates to both the quality of the heuristic and of the policy. We show empirically on the sliding-tile puzzle, Sokoban, and a puzzle from the commercial game 'The Witness' that PHS enables the rapid learning of both a policy and a heuristic function and compares favorably with A*, Weighted A*, Greedy Best-First Search, LevinTS, and PUCT in terms of number of problems solved and search time in all three domains tested.

1 Introduction

In this work[1] we are interested in tackling single-agent deterministic problems. This class of problems includes numerous real-world applications such as robotics, planning and pathfinding, computational biology (Edelkamp, Schroedl, and Koenig 2010), protein design (Allouche et al. 2019), and program synthesis (Cropper and Dumancic 2020).

AlphaGo (Silver et al. 2017), and descendants such as MuZero (Schrittwieser et al. 2020) combine a learned value function with a learned policy in the PUCT search algorithm (Rosin 2011; Kocsis and Szepesvári 2006), which is a Monte-Carlo tree search algorithm (Chang et al. 2005; Coulom 2007), to tackle stochastic and adversarial games with complete information, and also a few single-agent games. The policy guides the search locally by favoring the most promising children of a node, whereas the value function ranks paths globally, thus complementing each other. PUCT, based on UCT (Kocsis and Szepesvári 2006) and PUCB (Rosin 2011), is indeed designed for adversarial and stochastic games such as Go and Chess, and UCT and PUCB come with a guarantee that the value function converges to the true value function in the limit of exploration—however this guarantee may not hold when replacing actual rewards with an estimated value, as is done in the mentioned works.

Although these algorithms perform impressively well for some adversarial games, it is not clear whether the level of generality of PUCT makes it the best fit for difficult deterministic single-agent problems where a planning capability is necessary, such as the PSPACE-hard Sokoban problem (Culberson 1999). The more recent algorithm MuZero (Schrittwieser et al. 2020) adapts AlphaZero to single-agent Atari games, but these games are mostly reactive and MuZero performs poorly on games that require more planning like Montezuma's Revenge—although this may arguably pertain to MuZero needing to learn a model of the environment.

In the context of single-agent search the value function is known as a *heuristic* function and it estimates the cost-to-go from a given state to a solution state. McAleer et al. (2019) used MCTS to learn a heuristic function—but not a policy—to tackle the Rubik's cube, but later replaced MCTS entirely with weighted A* (Pohl 1970; Ebendt and Drechsler 2009), which is a variant of the A* algorithm (Hart, Nilsson, and Raphael 1968) that trades off solution quality for search time. They observe that "MCTS has relatively long runtimes and often produces solutions many moves longer than the length of a shortest path" (Agostinelli et al. 2019), and tackle a few more problems such as the sliding tile puzzle and Sokoban.

Levin Tree Search (LevinTS) (Orseau et al. 2018) uses a learned policy to guide its search in single-agent problems and comes with an upper bound on the number of search steps that accounts for the quality of the policy.

In this work we combine the policy-guided search of the LevinTS algorithm with the heuristic-guided search of A* in an algorithm we call Policy-guided Heuristic Search (PHS). PHS retains the upper bound of LevinTS—we also prove an almost matching lower bound—but we extend this guarantee to the use of a heuristic function, showing that an accurate heuristic can greatly reduce the number of search steps.

We compare our algorithm with several policy-guided and

[1]An extended version of this paper can be found at http://arxiv.org/abs/2103.11505.

heuristic search algorithms when learning is interleaved with search in the Bootstrap process (Jabbari Arfaee, Zilles, and Holte 2011): LevinTS uses a policy; A*, Weighted A* (WA*) and Greedy Best-First Search (GBFS) (Doran, Michie, and Kendall 1966) all use a heuristic, and PUCT uses both like PHS. We evaluate these algorithms on the 5×5 sliding-tile puzzle, Sokoban (Boxoban), and on a puzzle from the game 'The Witness'. Our results show that PHS performs well on all three domains tested, while every other algorithm tested performs poorly in at least one of these domains.

2 Notation and Background

Search algorithms solve single-agent problems by searching in the tree that defines the problem. Let $\mathcal{N}$ be the set of all nodes that can be part of such a tree. The root of the tree is denoted $n_0 \in \mathcal{N}$. For any node $n \in \mathcal{N}$, its set of children is $\mathcal{C}(n)$, its single parent is $\text{par}(n)$ (the root has no parent), its set of ancestors including n itself is $\text{anc}_*(n)$, and its set of descendants including n itself is $\text{desc}_*(n)$; its depth is $d(n)$ ($d(n_0) = 0$) and also define $d_0(n) = d(n) + 1$.

We say that a node is *expanded* when a search algorithm generates the set of children of the node. All search algorithms we consider are constrained to expand the root first and, subsequently, a node n can be expanded only if its parent $\text{par}(n)$ has already been expanded.

A problem is defined by a root node n_0, a non-negative loss function $\ell : \mathcal{N} \to [0, \infty]$, a set of solution nodes $\mathcal{N}_\mathcal{G} \subseteq \text{desc}_*(n_0)$, and a `state` function defined below. There is a predicate `is_solution`(n) available to the search algorithms to test whether $n \in \mathcal{N}_\mathcal{G}$, but it can be used on n only after $\text{par}(n)$ has been expanded (*i.e.*, n must have been generated). The loss $\ell(n)$ is incurred by the algorithm when expanding the node n. For simplicity of the formalism, we assume that a node which is tested positive with `is_solution` is implicitly expanded—thus incurring a loss—but has no children. The *path* loss $g(n)$ of a node n is the sum of the losses from the root to n, that is $g(n) = \sum_{n' \in \text{anc}_*(n)} \ell(n')$. For example, if the loss is one for any expanded node n, then $g(n) = d_0(n)$. We assume that no infinite path has finite path loss. For any search algorithm S, the *search* loss $L(S, n)$ is the sum of the individual losses $\ell(n')$ for all nodes n' that have been expanded by the algorithm S, up to and including n, and $L(S, n) = \infty$ if n is never expanded. For example, if the loss $\ell(n)$ is the time needed to expand node n, then $L(S, n)$ corresponds to the computation time of the whole search when reaching n.

A policy $\pi : \mathcal{N} \to [0, 1]$ is defined recursively for a child n' of a node n: $\pi(n') = \pi(n)\pi(n'|n)$ where the conditional probability $\pi(n'|n) \in [0, 1]$ is such that $\sum_{n' \in \mathcal{C}(n)} \pi(n'|n) \leq 1$, and $\pi(n_0) = 1$. Therefore, $\pi(n) = \prod_{n' \in \text{anc}_*(n) \setminus \{n_0\}} \pi(n'|\text{par}(n'))$.

Let $\mathcal{S} \subseteq \mathcal{N}$ be a set of 'canonical nodes'. The function `state` $: \mathcal{N} \to \mathcal{S}$ associates a node to a state (a canonical node), with the constraints that $\ell(n) = \ell(\texttt{state}(n))$, `is_solution`$(n)$ = `is_solution`$(\texttt{state}(n))$ and $\{\texttt{state}(n') : n' \in \mathcal{C}(n)\} = \mathcal{C}(\texttt{state}(n))$. Search algorithms may use the `state` function to avoid expanding nodes with the same states.

2.1 Background

The Best-First Search (BFS) search algorithm (Pearl 1984) (see Algorithm 1) expands nodes by increasing value, starting from the root and always expanding children only if their parent has been expanded already. It does not expand nodes whose states have been visited before, and returns the first solution node it expands.

The A* search algorithm (Hart, Nilsson, and Raphael 1968) uses both the function g and a heuristic function $h : \mathcal{N} \to [0, \infty]$. It uses BFS with the evaluation function $f(n) = g(n) + h(n)$. If the heuristic h is *admissible*, *i.e.*, $g(n) + h(n)$ is a lower bound on the cost of the least-g-cost solution node below n, then A* is guaranteed[2] to return a solution with minimal g-cost. Weighted A* (Ebendt and Drechsler 2009) is a variant of A* that uses the evaluation function $f_w(n) = g(n) + w \cdot h(n)$ and has the guarantee that the first solution found has a g-cost no more than a factor w of the minimum cost solution if h is admissible and $w \geq 1$.

LevinTS (Orseau et al. 2018) also uses the BFS algorithm, but with the evaluation function $f_\pi = d_0(n)/\pi(n)$ for a given policy π. LevinTS is guaranteed to expand no more than $d_0(n^*)/\pi(n^*)$ nodes until the first solution n^* is found, that is, with $\ell(\cdot) = 1$ for all nodes, $L(\text{LevinTS}, n^*) \leq d_0(n^*)/\pi(n^*)$. Since $d_0(n^*)$ is fixed, it shows that the better the policy π, the shorter the search time. Theorem 4 provides an almost-matching lower bound.

The PUCT algorithm (Silver et al. 2016) is not based on BFS, but on UCT (Kocsis and Szepesvári 2006), which learns a value function from rewards, and on PUCB (Rosin 2011), which additionally uses a policy prior. Both ingredients are used to determine which node to expand next. The PUCT formula depends on the current number of node expansions performed during search. This dependency prevents the algorithm from being able to use a priority queue which requires the node values to not change over time. Hence each time the algorithm performs a single node expansion, it has to go back to the root to potentially take a different path for the next expansion. In some cases, this additional cost can lead to a quadratic increase of the computation time compared to the number of node expansions. Although this search strategy can be effective in stochastic and adversarial environments, it is often wasteful for deterministic single-agent problems. The original UCT algorithm (Kocsis and Szepesvári 2006) has regret-style guarantees, but as noted by Orseau et al. (2018), these guarantees are rarely meaningful in deterministic single-agent problems where rewards are often 0 until a solution is found; moreover, these guarantees do not hold for modern implementations that replace the rewards and the rollouts with a learned value function.

2.2 Definition of the Search Problem

Our overarching objective is to design algorithms that, when given a set of unknown tasks, solve all of them as quickly as possible while starting with little to no knowledge about the tasks. That is, for K tasks, we want to devise an algorithm S that minimizes the total *search* loss

[2]Technically, this requires either that re-expansions are performed or that the heuristic is *consistent*.

$\sum_{k\in[K]} \min_{n^*\in\mathcal{N}_{\mathcal{G}_k}} L_k(S, n^*)$. Observe that this departs from the more traditional objective of finding solutions of minimum *path* loss for all tasks, that is, of minimizing $\sum_{k\in[K]} g(n_k^*)$ where n_k^* is the solution node found by the algorithm for task k. Hence, we do not require the solutions encountered by the search algorithms to be path-cost optimal.

To this end we embed our algorithms into the Bootstrap process (Jabbari Arfaee, Zilles, and Holte 2011), which iteratively runs a search algorithm with a bound on the number of node expansions (or running time) on a set of training tasks. The problems solved in a given iteration are used to update the parameters of a model encoding a heuristic function and/or a policy. The process is then repeated with the newly trained model, possibly after increasing the budget. This process does not use or generate a curriculum, but assumes that there are problems of varying difficulties.

Algorithm 1 The Policy-guided Heuristic Search algorithm (PHS), based on the Best-First Search algorithm (BFS). (Re-expansions are not performed.)

```
def PHS(n_0): return BFS(n_0, φ)

def BFS(n_0, evaluate):
  q = priority_queue(order_by=evaluate)
  q.insert(n_0)
  visited_states = {}
  while q is not empty:
    n = q.extract_min() # node of min value
    s = state(n)
    if s in visited_states:
      continue  # pruning
    visited_states += {s}
    incur_loss ℓ(n)
    if is_solution(n):
      return n
    # Node expansion
    for n' in children(n):
      q.insert(n', evaluate(n'))
  return False
```

3 Policy-guided Heuristic Search

For all our theoretical results, we assume that $\forall n \in \mathcal{N}$: state(n)=n to avoid having to deal with re-expansions.

We generalize LevinTS first by considering arbitrary non-negative losses $\ell(n)$ rather than enforcing $\ell(n) = 1$, and by introducing a *heuristic factor* $\eta(n) \geq 1$. Our algorithm Policy-guided Heuristic Search (PHS) simply calls BFS (see Algorithm 1) with the evaluation function φ defined as:

$$\varphi(n) = \eta(n)\frac{g(n)}{\pi(n)},$$

where g and π were defined in Section 2, and $\varphi(n) = \infty$ if $\pi(n) = 0$; Orseau et al. (2018) discuss how to prevent $\pi(n) = 0$. The factor $g(n)/\pi(n)$ can be viewed as an approximation of the search loss $L(\text{PHS}, n)$ when reaching n: $g(n)$ is the loss of the path from the root to n, and $1/\pi(n)$ plays a similar role to an importance sampling weight that rescales the (local) path loss to the (global) search loss. Note that paths with large probability π are preferred to be expanded before paths with small π-values. The purpose of the new heuristic factor $\eta(n)$ is to rescale the current estimate of the search loss $L(\cdot, n)$ to an estimate of the search loss $L(\cdot, n^*)$ at the least-cost solution node n^* that descends from n. This is similar to how $f(n)$ is an estimate of $g(n^*)$ in A*. If both $\eta(\cdot)$ and $\ell(\cdot)$ are 1 everywhere, PHS reduces to LevinTS.

Although for LevinTS f_π is monotone non-decreasing from parent to child (Orseau et al. 2018), this may not be the case anymore for φ due to η. Thus, for the sake of the analysis we define the monotone non-decreasing evaluation function $\varphi^+(n) = \max_{n'\in\text{anc}_*(n)} \varphi(n')$. Since in BFS nodes are expanded in increasing order of their value, and a node n cannot be expanded before any of its ancestors, it is guaranteed in PHS that a node n is expanded *before* any other node n' with $\varphi^+(n') > \varphi^+(n)$.

Define $\eta^+(n) = \varphi^+(n)\frac{\pi(n)}{g(n)} \geq \eta(n)$, and $\eta^+(n) = \eta(n)$ if $g(n) = 0$, such that

$$\varphi^+(n) = \eta^+(n)\frac{g(n)}{\pi(n)}.$$

Note that even if we ensure that $\eta(n^*) = 1$ for any solution node $n^* \in \mathcal{N}_{\mathcal{G}}$, in general $\varphi^+(n^*)$ may still be larger than $\varphi(n^*)$, that is, we may have $\eta^+(n^*) > 1$.

We now state our main result and explain it below. Define the set of nodes $\mathcal{N}_\varphi(n) = \{n' \in \text{desc}_*(n_0) : \varphi^+(n') \leq \varphi^+(n)\}$ of value at most $\varphi^+(n)$ and its set of leaves $\mathcal{L}_\varphi(n) = \{n' \in \mathcal{N}_\varphi(n) : \mathcal{C}(n') \cap \mathcal{N}_\varphi(n) = \emptyset\}$ that is, the set of nodes in $\mathcal{N}_\varphi(n)$ that do not have any children in this set.

Theorem 1 (PHS upper bound)**.** For any non-negative loss function ℓ, for any set of solution nodes $\mathcal{N}_{\mathcal{G}} \subseteq \text{desc}_*(n_0)$, and for any given policy π and any given heuristic factor $\eta(\cdot) \geq 1$, PHS returns a solution node $n^* \in \text{argmin}_{n^*\in\mathcal{N}_{\mathcal{G}}} \varphi^+(n^*)$ and the search loss is bounded by

$$L(\text{PHS}, n^*) \leq \frac{g(n^*)}{\pi(n^*)}\eta^+(n^*) \sum_{n\in\mathcal{L}_\varphi(n^*)} \frac{\pi(n)}{\eta^+(n)}. \quad (1)$$

Proof. Slightly abusing notation, for a set of nodes $\mathcal{N}'$, define $L(\mathcal{N}') = \sum_{n\in\mathcal{N}'} \ell(n)$ to be the cumulative loss over the nodes in $\mathcal{N}'$. Since φ^+ is non-decreasing from parent to child, $\mathcal{N}_\varphi(n)$ forms a tree rooted in n_0 and therefore all the nodes in $\mathcal{N}_\varphi(n)$ are expanded by BFS(n_0, φ) before any other node not in $\mathcal{N}_\varphi(n)$. Therefore, $L(\text{PHS}, n) \leq L(\mathcal{N}_\varphi(n))$. Then,

$$\begin{aligned} L(\mathcal{N}_\varphi(n)) &= L\Big(\bigcup_{n'\in\mathcal{L}_\varphi(n)} \text{anc}_*(n')\Big) \\ &\leq \sum_{n'\in\mathcal{L}_\varphi(n)} L(\text{anc}_*(n')) \\ &= \sum_{n'\in\mathcal{L}_\varphi(n)} g(n') \leq \varphi^+(n) \sum_{n'\in\mathcal{L}_\varphi(n)} \frac{\pi(n')}{\eta^+(n')} \end{aligned}$$

where the last inequality follows from $\eta^+(n')\frac{g(n')}{\pi(n')} = \varphi^+(n') \leq \varphi^+(n)$. Finally, since this is true for any n, the result holds for the returned $n^* \in \mathcal{N}_{\mathcal{G}}$. □

We can derive a first simpler result:

Corollary 2 (PHS upper bound with no heuristic)**.** From Theorem 1, if furthermore $\forall n, \eta(n) = 1$ then

$$L(\text{PHS}, n^*) \leq \frac{g(n^*)}{\pi(n^*)}\,. \tag{2}$$

Proof. Follows from Theorem 1 and $\sum_{n' \in \mathcal{L}_\varphi(n)} \pi(n') \leq 1$ for all n (Orseau et al. 2018, Theorem 3). □

The bound in Eq. (2) is similar to the bound provided for LevinTS (and equal when $\ell(n) = 1$ everywhere) and shows the effect of the quality of the policy on the cumulative loss during the search: Since necessarily $L(\text{PHS}, n^*) \geq g(n^*)$, the bound says that the search becomes more efficient as $\pi(n^*)$ gets closer to 1. Conversely, with an uninformed policy π the probability decreases exponentially with the depth, which means that the search becomes essentially a breadth-first search. The bound in Eq. (1) furthermore shows the effect of the heuristic function where the additional factor can be interpreted as the ratio of the heuristic value $\eta^+(n^*)$ at the solution to the (preferably larger) average heuristic value at the leaves of the search tree when reaching n^*. If the heuristic is good, then this bound can substantially improve upon Eq. (2)—but it can also degrade with a poor heuristic.

Example 3. Consider a binary tree where the single solution node n^* is placed at random at depth d. Assume that $\pi(n'|n) = 1/2$ for both children n' of any node n. Assume that $\eta(n) = \infty$ for all nodes except $\eta(n) = 1$ for the nodes on the path from the root to n^*, which makes PHS expand only the $d + 1$ nodes on the path from the root to n^*. Take $\ell(n) = 1$ for all nodes. Then Eq. (2) tells us that the search loss (which is here the number of expanded nodes) is bounded by $(d + 1)2^{d+1}$, which is correct but rather loose. By taking the (very informative) heuristic information into account, we have $\sum_{n \in \mathcal{L}_\varphi(n^*)} \frac{\pi(n)}{\eta^+(n)} = \pi(n^*)/\eta(n^*)$ and thus Eq. (1) tells us that the search loss is bounded by $g(n^*) = d + 1$, which is tight. △

The following lower bound is close to the bound of Eq. (2) in general, and applies to any algorithm, whether it makes use of the policy or not. First, we say that a policy π is *proper* if for all nodes n, $\sum_{n' \in \mathcal{C}(n)} \pi(n'|n) = 1$.

Theorem 4 (Lower bound)**.** For every proper policy π and non-negative loss function ℓ such that $\ell(n_0) = 0$, for every search algorithm S—that first expands the root and subsequently expands only children of nodes that have been expanded—there are trees rooted in n_0 and sets of solution nodes $\mathcal{N}_\mathcal{G}$ such that

$$\min_{n^* \in \mathcal{N}_\mathcal{G}} L(S, n^*) \geq \frac{g(\hat{n}^*)}{\pi(n^*)}, \quad \text{with } \hat{n}^* = \text{par}(\text{par}(n^*))\,.$$

Proof. Consider the following infinite tree: The root n_0 has m children, $n_{1,1}, n_{2,1} \ldots n_{m,1}$ and each child $n_{i,1}$ is assigned an arbitrary conditional probability $\pi(n_{i,1}|n_0) \geq 0$ such that $\sum_{i \in [m]} \pi(n_{i,1}|n_0) = 1$. Each node $n_{i,j}$ has a single child $n_{i,j+1}$ with $\pi(n_{i,j+1}|n_{i,j}) = 1$, and is assigned an arbitrary loss $\ell(n_{i,j}) \geq 0$. Observe that $\pi(n_{i,j}) = \pi(n_{i,1}|n_0)$.

Now let the given search algorithm expand nodes in any order, as long as parents are always expanded before their children, until at least one node is expanded in each branch with positive probability—if this is not met then the bound holds trivially. Then stop the search after any finite number of steps. Let $\hat{n}_i$ be the last expanded node in each branch $i \in [m]$. Then pick the node $\hat{n}^*$ among the $\hat{n}_i$ with smallest $g(\hat{n}_i)/\pi(\hat{n}_i)$. Since this node $\hat{n}^*$ has been expanded, its unique child $\hat{n}'$ may have already been tested for solution, but not its grand-child since $\hat{n}'$ has not yet been expanded. So we set n^* to be the unique child of $\hat{n}'$, and set $\mathcal{N}_\mathcal{G} = \{n^*\}$. For each branch i we have $g(\hat{n}_i)/\pi(\hat{n}_i) \geq g(\hat{n}^*)/\pi(\hat{n}^*)$. Therefore, recalling that the policy is proper, the cumulative loss before testing if n^* is a solution is at least

$$\sum_{i \in [m]} g(\hat{n}_i) \geq \sum_{i \in [m]} \pi(\hat{n}_i) \frac{g(\hat{n}^*)}{\pi(\hat{n}^*)} = \frac{g(\hat{n}^*)}{\pi(\hat{n}^*)} = \frac{g(\hat{n}^*)}{\pi(n^*)}\,. \square$$

3.1 Admissible Heuristics

We say that η is *PHS-admissible* (by similarity to admissibility of heuristic functions for A*) if for all nodes n, for all solution nodes n^* below n (*i.e.*, $n^* \in \text{desc}_*(n) \cap \mathcal{N}_\mathcal{G}$), we have $\varphi(n) \leq \varphi(n^*)$ and $\eta(n^*) = 1$, that is, $\varphi(n^*) = g(n^*)/\pi(n^*)$. This means that $\varphi(n)$ always underestimates the cost of any descendant solution. This ensures that for solution nodes, $\varphi^+(n^*) = \varphi(n^*) = g(n^*)/\pi(n^*)$. Note that we may still not have $\varphi^+(n) = \varphi(n)$ for non-solution nodes. Also observe that taking $\eta(n) = 1$ for all n is admissible, but not informed, similarly to $h(n) = 0$ in A*.

Hence, *ideally*, if $\eta(n) = \infty$ when $\text{desc}_*(n) \cap \mathcal{N}_\mathcal{G} = \emptyset$,

$$\text{and } \eta(n) = \min_{n* \in \text{desc}_*(n) \cap \mathcal{N}_\mathcal{G}} \frac{g(n^*)/\pi(n^*)}{g(n)/\pi(n)},$$
$$\text{then } \varphi(n) = \min_{n* \in \text{desc}_*(n) \cap \mathcal{N}_\mathcal{G}} g(n^*)/\pi(n^*)$$

and thus, similarly to A*, PHS does not expand any node which does not lead to a solution node of minimal φ-value. When the heuristic is PHS-admissible but not necessarily ideal, we provide a refined bound of Eq. (1):

Corollary 5 (Admissible upper bound)**.** If η is PHS-admissible, then the cumulative loss incurred by PHS before returning a solution node $n^* \in \text{argmin}_{n^* \in \mathcal{N}_\mathcal{G}} \varphi^+(n^*)$ is upper bounded by

$$L(\text{PHS}, n^*) \leq \frac{g(n^*)}{\pi(n^*)} \underbrace{\sum_{n \in \mathcal{L}_\varphi(n^*)} \frac{\pi(n)}{\eta^+(n)}}_{\Sigma}\,. \tag{3}$$

Proof. Follows from Eq. (1) with $\varphi^+(n^*) = \varphi(n^*)$ due to the PHS-admissibility of η. □

Corollary 5 offers a better insight into the utility of a heuristic factor η, in particular when it is PHS-admissible. First, observe that $\sum_{n' \in \mathcal{L}_\varphi(n)} \pi(n') \leq 1$, which means that the sum term Σ can be interpreted as an average. Second, since $\eta^+(\cdot) \geq 1$, then necessarily $\Sigma \leq 1$, which means that Corollary 5 is a strict improvement over Eq. (2). Σ can thus be read as the average search reduction factor at the leaves of

the search tree when finding node n^*. Corollary 5 shows that using a PHS-admissible heuristic factor η can help the search, on top of the help that can be obtained by using a good policy. In light of this, we can now interpret $\eta^+(n^*)$ in Eq. (1) as an *excess estimate* for inadmissible η, and a trade-off appears between this excess and the potential gain in the Σ term by the heuristic. Interestingly, this trade-off disappears when the heuristic factor is PHS-admissible, that is, the bounds suggest that using a PHS-admissible heuristic is essentially 'free.'

A number of traditional problems have readily available admissible heuristics for A*. We show that these can be used in PHS too to define a PHS-admissible η. Let h be a heuristic for A*. Making the dependency on h explicit, define

$$\eta_h(n) = \frac{g(n)+h(n)}{g(n)}\,, \quad \text{then } \varphi_h(n) = \frac{g(n)+h(n)}{\pi(n)}\,.$$

Theorem 6 (A*-admissible to PHS-admissible)**.** If h is an admissible heuristic for A*, then η_h is PHS-admissible.

Proof. Since h is admissible for A*, we have $g(n)+h(n) \leq g(n^*)$ for any solution node n^* descending from n. Hence $\varphi_h(n) \leq g(n^*)/\pi(n) \leq g(n^*)/\pi(n^*) = \varphi_h(n^*)$ and thus η_h is PHS-admissible. □

Define $h^+(n) = \pi(n)\max_{n'\in \text{anc}_*(n)} \varphi_h(n') - g(n) \geq h(n)$ which is such that $\varphi_h^+(n) = \max_{n'\in\text{anc}_*(n)}\varphi_h(n')$ is monotone non-decreasing. Then $\eta_h^+(n) = 1 + h^+(n)/g(n)$.

Corollary 7 (Admissible upper bound with h)**.** Given a heuristic function h, if $\eta = \eta_h$ is PHS-admissible, then the cumulative loss incurred before returning a solution node $n^* \in \text{argmin}_{n^*\in\mathcal{N}_\mathcal{G}}\,\varphi^+(n^*)$ is upper bounded by

$$L(\text{PHS}, n^*) \leq \frac{g(n^*)}{\pi(n^*)} \sum_{n\in\mathcal{L}_\varphi(n^*)} \frac{\pi(n)}{1+h^+(n)/g(n)}\,.$$

Proof. Follows by Corollary 5 and the definition of η_h^+. □

Corollary 7 shows that the larger $h(n)$ (while remaining admissible), the smaller the sum, and thus the smaller the bound. A well tuned heuristic can help reduce the cumulative loss by a large factor compared to the policy alone.

We call PHS$_h$ the variant of PHS that takes an A*-like heuristic function h and uses $\eta = \eta_h$.

3.2 A More Aggressive Use of Heuristics

Similarly to A*, the purpose of the heuristic factor is to estimate the g-cost of the least-cost descendant solution node. But even when h is admissible and accurate, η_h is often not an accurate estimate of the cumulative loss at n^* due to missing the ratio $\pi(n)/\pi(n^*)$—and quite often $\pi(n^*) \ll \pi(n)$. Hence if h is the only known heuristic information, we propose an estimate $\hat{\eta}_t$ that should be substantially more accurate if conditional probabilities and losses are sufficiently regular on the path to the solution, by approximating $\pi(n^*)$ as $[\pi(n)^{1/g(n)}]^{g(n)+h(n)}$: Take $g(n) = d(n)$ for intuition, then $\pi(n)^{1/g(n)} = p$ is roughly the average conditional probability along the path from the root to n, and thus $p^{g(n)+h(n)}$ is an estimate of the probability at depth $d(n^*)$, that is, an estimate of $\pi(n^*)$. This gives

$$\hat{\eta}_h(n) = \frac{1+h(n)/g(n)}{\pi(n)^{h(n)/g(n)}}\,,\ \hat{\varphi}_h(n) = \frac{g(n)+h(n)}{\pi(n)^{1+h(n)/g(n)}} \quad (4)$$

so that $\hat{\varphi}_h(n)$ is an estimate of $g(n^*)/\pi(n^*)$. The drawback of $\hat{\eta}_h$ is that it may not be PHS-admissible anymore, so while Theorem 1 still applies, Corollary 5 does not.

We call PHS* the variant of PHS that defines η as in Eq. (4) based on some given (supposedly approximately admissible) heuristic function h.

4 Learning the Policy

We consider K tasks, and quantities indexed by $k \in [K]$ have the same meaning as before, but for task k. We assume that the policy π_θ is differentiable w.r.t. its parameters $\theta \in \Theta$. Ideally, we want to optimize the policy so as to minimize the total search loss, *i.e.*, the optimal parameters of the policy are

$$\theta^* = \operatorname*{argmin}_{\theta\in\Theta} \sum_{k\in[K]} \min_{n^*\in\mathcal{N}_{\mathcal{G}_k}} L_k(\text{PHS}_\theta, n^*)\,.$$

Unfortunately, even if we assume the existence of a differentiable close approximation $\tilde{L}_k$ of L_k, the gradient of the sum with respect to θ can usually not be obtained. Instead the upper bound in Eq. (1) can be used as a surrogate loss function. However, it is not ideally suited for optimization due to the dependence on the set of leaves $\mathcal{L}_\varphi(n^*)$. Hence we make a crude simplification of Eq. (1) and assume that for task k, $L_k(\text{PHS}_\theta, n_k^*) \approx c_k g(n_k^*)/\pi_\theta(n_k^*)$ for some a priori unknown constant c_k assumed independent of the parameters θ of the policy: indeed the gradient of the term in Eq. (1) that c_k replaces should usually be small since $\pi_\theta(n)$ is often an exponentially small quantity with the search depth and thus $\nabla_\theta\pi_\theta(n) \approx 0$. Then, since $\mathrm{d}f(x)/\mathrm{d}x = f(x)\mathrm{d}\log f(x)/\mathrm{d}x$,

$$\begin{aligned}\nabla_\theta \tilde{L}_k(\text{PHS}_\theta, n_k^*) &= \tilde{L}_k(\text{PHS}, n_k^*)\nabla_\theta \log \tilde{L}_k(\text{PHS}_\theta, n_k^*) \\ &\approx \tilde{L}_k(\text{PHS}_\theta, n_k^*)\nabla_\theta \log\Big(c_k \frac{g(n_k^*)}{\pi_\theta(n_k^*)}\Big) \\ &\approx L_k(\text{PHS}_\theta, n_k^*)\nabla_\theta \log \frac{1}{\pi_\theta(n_k^*)}\,. \quad (5)\end{aligned}$$

Note that $L_k(\text{PHS}_\theta, n_k^*)$ can be calculated during the search, and that the gradient does not depend explicitly on the heuristic function h, which is learned separately. In the experiments, we use this form to optimize the policy for PHS and LevinTS.

5 Experiments

We use $\ell(\cdot) = 1$ everywhere, so the search loss L corresponds to the number of node expansions. We test the algorithms A*, GBFS, WA* (w=1.5), PUCT (c=1), PHS$_h$ (using $\eta = \eta_h$) and PHS* (using $\eta = \hat{\eta}_h$), and LevinTS. Each algorithm uses one neural network to model the policy and/or the heuristic to be trained on the problems it manages to solve. For PUCT, we normalize the Q-values (Schrittwieser et al. 2020), and use a virtual loss (Chaslot, Winands, and van den Herik 2008)

of unit increment with the following selection rule for a child n' of a node n,

$$\bar{h}(n') = (h(n') + \text{virtual\_loss}(n') - h_{\min})/(h_{\max} - h_{\min})$$

$$\text{PUCT}(n'; n) = \bar{h}(n) - c\pi(n'|n)\frac{\sqrt{\sum_{n'' \in \mathcal{C}(n)} N(n'')}}{1 + N(n')}$$

where $N(n)$ is the number of times the node n has been visited, c is a constant set to 1 in the experiments, and keep in mind that h corresponds to losses—hence the negative sign. The node n' with minimum PUCT value is selected. The virtual loss allows for evaluating the nodes in batch: we first collect 32 nodes for evaluation using the PUCT rule and only then evaluate all nodes in batch with the neural network. The virtual loss allows us to sample different paths of the MCTS tree to be evaluated. Similarly, for the BFS-based algorithms, 32 nodes are evaluated in batch with the neural network before insertion in the priority queue (Agostinelli et al. 2019). This batching speeds up the search for all algorithms.

Since we want to assess the cooperation of search and learning capabilities of the different algorithms without using domain-specific knowledge, our experiments are not directly comparable with domain-specific solvers (see Pereira et al. (2016) for Sokoban). In particular, no intermediate reward is provided, by contrast to Orseau et al. (2018) for example.

5.1 Domains

Sokoban (10×10) Sokoban is a PSPACE-hard grid-world puzzle where the player controls an avatar who pushes boxes to particular spots (boxes cannot be pulled). We use the first 50 000 problems training problems and the provided 1 000 test problems from Boxoban (Guez et al. 2018).

The Witness (4×4) The Witness domain is a NP-complete puzzle extracted from the video game of the same name (Abel et al. 2020) and consists in finding a path on a 2D grid that separates cells of different colors.[3]

Sliding Tile Puzzle (5×5) The sliding tile puzzle is a traditional benchmark in the heuristic search literature where heuristic functions can be very effective (Korf 1985; Felner, Korf, and Hanan 2004). The training set is generated with random walks from the 5×5 solution state with walks of lengths between 50 and 1 000 steps—this is a difficult training set as there are no very easy problems. Test problems are generated randomly and unsolvable problems are filtered out by a parity check; note that this is like scrambling infinitely often, which makes the test problems often harder than the training ones.

5.2 Training and Testing

Each search algorithm start with a uniform policy and/or an uninformed heuristic and follow a variant of the Bootstrap process (Ernandes and Gori 2004; Jabbari Arfaee, Zilles, and Holte 2011): All problems are attempted with an initial search step budget (2 000 for Witness and Sokoban, 7 000 for the sliding tile puzzle) and the search algorithm may solve some of these problems. After 32 attempted problems, a parameter update pass of the models is performed, using as many data points as the lengths of the solutions of the solved problems among the 32. If no new problem has been solved at the end of the whole Bootstrap iteration, the budget is doubled. Then the next iteration begins with all problems again. The process terminates when the total budget of 7 days is spent (wall time, no GPU). We use the mean squared error (MSE) loss for learning the heuristic functions: For a found solution node n^* the loss for node $n \in \text{anc}_*(n^*)$ is $[(d(n^*) - d(n)) - h(n)]^2$, where $h(n)$ is the output of the network. Note that this loss function *tends* to make the heuristic admissible. The cross-entropy loss is used to learn the policy for PUCT. The policy for LevinTS and our PHS variants are based on the approximate gradient of the $\tilde{L}_k$ loss (see Section 4).

Testing follows the same process as training (with a total computation time of 2 days), except that parameters are not updated, and the budget is doubled unconditionally at each new Bootstrap iteration. The test problems are not used during training. Each algorithm is trained 5 times with random initialization of the networks. For fairness, we test each algorithm using its trained network that allowed the search algorithm to solve the largest number of training problems.

5.3 Results

The learning curves and test results are in Fig. 1 and Table 1.

Sokoban PHS$_h$ and PHS* obtain the best results on this domain, showing the effectiveness of combining a policy and a heuristic function with BFS's efficient use of the priority queue. LevinTS and WA* follow closely. A* takes significantly more time but finds shorter solutions, as expected. By contrast to the other algorithms, GBFS is entirely dependent on the quality of the heuristic function and cannot compensate an average-quality heuristic with a complete search (A* and WA* also use the g-cost to help drive the search). After the training period, PUCT could not yet learn a good policy and a good heuristic from scratch: PUCT is designed for the larger class of stochastic and adversarial domains and takes a large amount of time just to expand a single new node. Some variants of PUCT commit to an action after a fixed number of search steps (*e.g.*, Racanière et al. (2017)); although once a good value function has been learned this may help with search time, it also makes the search incomplete, which is not suitable when combining search and learning from scratch. Agostinelli et al. (2019) report better results than ours for WA* (about 1 050 000 expansions in total on the test set), but their network has more than 30 times as many parameters, and they use 18 times as many training problems, while also using a backward model to generate a curriculum. For LevinTS, Orseau et al. (2018) reported 5 000 000 expansions, using a larger network, 10 times as many training steps and intermediate rewards. Our improved results for LevinTS are likely due to the cooperation of search and learning during training, allowing to gather gradients for harder problems.

The Witness This domain appears to be difficult for learning a good heuristic function, with the policy-guided BFS-

[3]Datasets and code are at https://github.com/levilelis/h-levin.

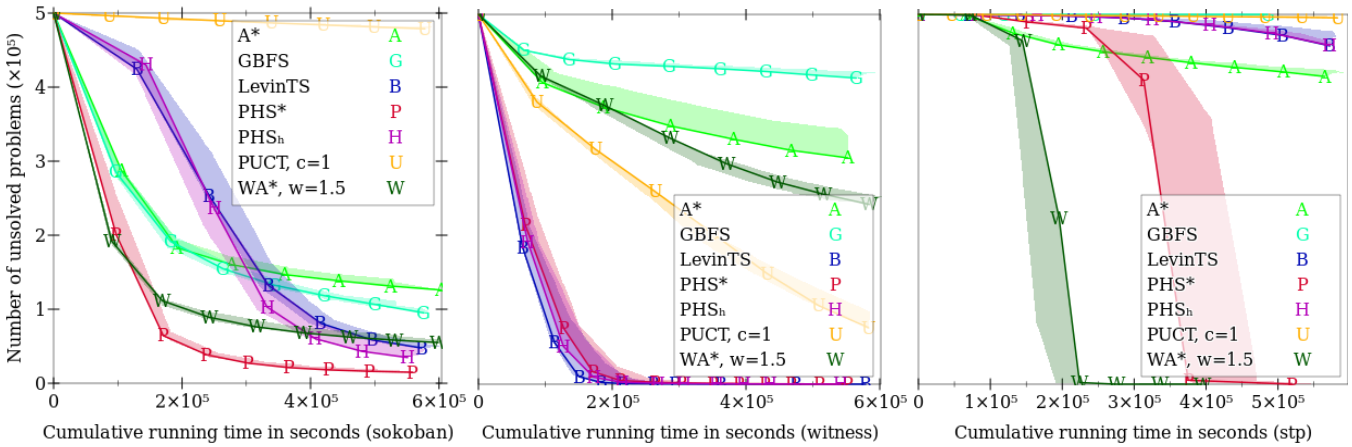

Figure 1: Learning curves using the Bootstrap process. Each point is a Bootstrap iteration. Lines correspond to the training runs with the least remaining unsolved problems at the end of the run. All 5 training runs per algorithm lie within the colored areas.

based algorithms being the clear winners. Even for PHS, the heuristic function does not seem to help compared to LevinTS; it does not hurt much either though, showing the robustness of PHS. PUCT and WA* learned slowly during training but still manage to solve (almost) all test problems, at the price of many expansions. GBFS performs poorly because it relies exclusively on the quality of the heuristic—by contrast, A* and WA* use the g-cost to perform a systematic search when the heuristic is not informative.

Sliding Tile Puzzle Only WA* and PHS* manage to solve all test problems. PHS$_h$ seems to not be using the heuristic function to its full extent, by contrast to PHS*. The trend in the learning curves (see Fig. 1) suggests that with more training LevinTS, PHS$_h$, and A* would achieve better results, but both GBFS and PUCT seem to be stuck. Possibly the training set is too difficult for these algorithms and adding easier problems or an initial heuristic could help.

6 Conclusion

We proposed a new algorithm called PHS that extends the policy-based LevinTS to using general non-negative loss functions and a heuristic function. We provided theoretical results relating the *search* loss with the quality of both the policy and the heuristic. If the provided heuristic function is PHS-admissible, a strictly better upper bound can be shown for PHS than for LevinTS. In particular, an admissible heuristic for A* can be turned into a PHS-admissible heuristic, leading to the variant PHS$_h$. We also provided a lower bound based on the information carried by the policy, that applies to any search algorithm. The more aggressive variant PHS* is the only algorithm which consistently solves all test problems in the three domains tested. It would be useful to derive more specific bounds showing when PHS* is expected to work strictly better than PHS$_h$. In this paper, the learned heuristic corresponds to the distance to the solution as for A*, but it may be better to directly learn the heuristic η to estimate the actual *search* loss at the solution. This may however introduce some difficulties since the heuristic function to learn would now depend on the policy, making learning possibly less stable.

Alg.	Solved	Length	Expansions	Time (s)
Sokoban 10×10 (test)				
PUCT, c=1	229	*24.9*	*10 021.3*	*39.7*
GBFS	914	*36.4*	*5 040.0*	*34.0*
A*	995	**32.7**	*8 696.8*	*61.7*
WA*, w=1.5	**1 000**	34.5	3 729.1	25.5
LevinTS	**1 000**	40.1	2 640.4	19.5
PHS$_h$	**1 000**	39.1	2 130.4	18.6
PHS*	**1 000**	37.6	**1 522.1**	**11.3**
The Witness 4×4 (test)				
GBFS	290	*13.3*	*10 127.9*	*44.6*
A*	878	*13.6*	*9 022.3*	*53.9*
WA*, w=1.5	999	**14.6**	*18 345.2*	*71.5*
PUCT, c=1	**1 000**	15.4	4 212.1	23.6
PHS*	**1 000**	15.0	781.5	5.4
LevinTS	**1 000**	14.8	520.2	3.2
PHS$_h$	**1 000**	15.0	**408.1**	**3.0**
Sliding Tile Puzzle 5×5 (test)				
GBFS	0	—	—	—
PUCT, c=1	0	—	—	—
A*	3	*87.3*	*34 146.3*	*27.2*
PHS$_h$	4	*119.5*	*58 692.0*	*55.3*
LevinTS	9	*145.1*	*39 005.6*	*31.1*
PHS*	**1 000**	224.0	2 867.2	2.8
WA*, w=1.5	**1 000**	**129.8**	**1 989.8**	**1.6**

Table 1: Results on the tests sets. Lengths are the solution depths, and the last three columns are averaged over solved problems only; hence numbers such as *123* cannot be properly compared with.

Acknowledgements

We would like to thank Tor Lattimore, Marc Lanctot, Michael Bowling, Ankit Anand, Théophane Weber, Joel Veness and the AAAI reviewers for their feedback and helpful comments. This research was enabled by Compute Canada (www.computecanada.ca) and partially funded by Canada's CIFAR AI Chairs program.

References

Abel, Z.; Bosboom, J.; Coulombe, M. J.; Demaine, E. D.; Hamilton, L.; Hesterberg, A.; Kopinsky, J.; Lynch, J.; Rudoy, M.; and Thielen, C. 2020. Who witnesses The Witness? Finding witnesses in The Witness is hard and sometimes impossible. *Theor. Comput. Sci.* 839: 41–102.

Agostinelli, F.; McAleer, S.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence* 1.

Allouche, D.; Barbe, S.; De Givry, S.; Katsirelos, G.; Lebbah, Y.; Loudni, S.; Ouali, A.; Schiex, T.; Simoncini, D.; and Zytnicki, M. 2019. Cost Function Networks to Solve Large Computational Protein Design Problems. In *Operations Research and Simulation in healthcare*. Springer.

Chang, H. S.; Fu, M. C.; Hu, J.; and Marcus, S. I. 2005. An Adaptive Sampling Algorithm for Solving Markov Decision Processes. *Operations Research* 53(1): 126–139.

Chaslot, G. M. J. B.; Winands, M. H. M.; and van den Herik, H. J. 2008. Parallel Monte-Carlo Tree Search. In *Computers and Games*, 60–71. Springer Berlin Heidelberg.

Coulom, R. 2007. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*, 72–83. Springer Berlin Heidelberg.

Cropper, A.; and Dumancic, S. 2020. Learning Large Logic Programs By Going Beyond Entailment. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, 2073–2079. ijcai.org.

Culberson, J. C. 1999. Sokoban is PSPACE-Complete. In *Fun With Algorithms*, 65–76.

Doran, J. E.; Michie, D.; and Kendall, D. G. 1966. Experiments with the Graph Traverser program. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 294(1437): 235–259.

Ebendt, R.; and Drechsler, R. 2009. Weighted A$*$ search – unifying view and application. *Artificial Intelligence* 173(14): 1310 – 1342.

Edelkamp, S.; Schroedl, S.; and Koenig, S. 2010. *Heuristic Search: Theory and Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 0123725127.

Ernandes, M.; and Gori, M. 2004. Likely-Admissible and Sub-Symbolic Heuristics. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, 613–617. IOS Press.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive Pattern Database Heuristics. *Journal of Artificial Intelligence Research* 22: 279–318.

Guez, A.; Mirza, M.; Gregor, K.; Kabra, R.; Racaniere, S.; Weber, T.; Raposo, D.; Santoro, A.; Orseau, L.; Eccles, T.; Wayne, G.; Silver, D.; Lillicrap, T.; and Valdes, V. 2018. An investigation of Model-free planning: boxoban levels. https://github.com/deepmind/boxoban-levels/, 14 Dec 2018.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2): 100–107.

Jabbari Arfaee, S.; Zilles, S.; and Holte, R. C. 2011. Learning heuristic functions for large state spaces. *Artificial Intelligence* 175(16): 2075–2098.

Kocsis, L.; and Szepesvári, C. 2006. Bandit Based Monte-Carlo Planning. In *ECML*, 282–293. Springer Berlin Heidelberg.

Korf, R. E. 1985. Depth-first iterative-deepening. *Artificial Intelligence* 27(1): 97 – 109.

McAleer, S.; Agostinelli, F.; Shmakov, A.; and Baldi, P. 2019. Solving the Rubik's Cube with Approximate Policy Iteration. In *International Conference on Learning Representations (ICRL)*.

Orseau, L.; Lelis, L.; Lattimore, T.; and Weber, T. 2018. Single-Agent Policy Tree Search With Guarantees. In *Advances in Neural Information Processing Systems 31*, 3201–3211. Curran Associates, Inc.

Pearl, J. 1984. *Heuristics - intelligent search strategies for computer problem solving*. Addison-Wesley series in artificial intelligence. Addison-Wesley. ISBN 978-0-201-05594-8.

Pereira, A. G.; Holte, R.; Schaeffer, J.; Buriol, L. S.; and Ritt, M. 2016. Improved Heuristic and Tie-Breaking for Optimally Solving Sokoban. In *International Joint Conference on Artificial Intelligence*.

Pohl, I. 1970. Heuristic search viewed as path finding in a graph. *Artificial Intelligence* 1(3): 193 – 204.

Racanière, S.; Weber, T.; Reichert, D.; Buesing, L.; Guez, A.; Jimenez Rezende, D.; Puigdomènech Badia, A.; Vinyals, O.; Heess, N.; Li, Y.; Pascanu, R.; Battaglia, P.; Hassabis, D.; Silver, D.; and Wierstra, D. 2017. Imagination-Augmented Agents for Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems 30*, 5690–5701. Curran Associates, Inc.

Rosin, C. D. 2011. Multi-Armed Bandits with Episode Context. *Annals of Mathematics and Artificial Intelligence* 61(3): 203–230.

Schrittwieser, J.; Antonoglou, I.; Hubert, T.; Simonyan, K.; Sifre, L.; Schmitt, S.; Guez, A.; Lockhart, E.; Hassabis, D.; Graepel, T.; Lillicrap, T.; and Silver, D. 2020. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature* 588(7839): 604–609.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529(7587): 484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T. P.; Simonyan, K.; and Hassabis, D. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *CoRR* abs/1712.01815.