Journal of Machine Learning Research 27 (2026) 1-64 Submitted 9/25; Revised 3/26; Published 4/26

Learning to Play Two-Player Perfect-Information Games without Knowledge

Quentin Cohen-Solal, Fixed-term researcher QUENTIN.COHEN-SOLAL@DAUPHINE.PSL.EU
LAMSADE, Université Paris Dauphine - PSL, CNRS
Place de Lattre de Tassigny, Paris, 75016, France
and
CRIL, Univ. Artois and CNRS
Rue Jean Souvraz SP 18, Lens, F-62300, France
and
Normandie University, UNICAEN, CNRS, GREYC
6 Boulevard du Maréchal Juin, Caen, 14 000, France

Editor: George Konidaris

Abstract

This paper introduces a set of techniques for learning game state evaluation functions through reinforcement learning. First, we generalize tree bootstrapping, i.e. learning the values of states encountered during search rather than restricting updates to states observed during matches, to the setting of reinforcement learning with non-linear function approximation. Second, we modifies Unbounded Best-First Minimax by extending best action sequences to terminal states. Third, we replace the traditional binary game outcome $+1/-1$ with richer reinforcement signals, including quick wins, delayed losses, and scoring. Fourth, we propose a completion mechanism that exploits state resolution. Finally, we introduce a novel action-selection distribution, referred to as the ordinal distribution.

Experimental results show that each of these techniques contributes to substantial improvements in playing strength. We integrate them into a unified algorithm, Athénan, and compare it against ExIt, a leading self-play reinforcement learning approach without prior knowledge. Our results demonstrate that Athénan consistently outperforms ExIt.

We further evaluate Athénan on the games Hex, Othello, and Arimaa, where it surpasses state-of-the-art performance without relying on domain-specific knowledge. In addition, we consider the single-player game Morpion Solitaire, in which Athénan again reaches state-of-the-art results under the same constraint.

Overall, these results show that reinforcement learning, when combined with the proposed techniques, can achieve state-of-the-art performance across a diverse range of games without the need for handcrafted heuristics or expert knowledge.

Keywords: Reinforcement Learning, Learning without Human Knowledge, Minimax, Search Algorithms, Games

1. Introduction

One of the most difficult tasks in artificial intelligence is the *sequential decision making problem* (Littman, 1996), whose applications include robotics and games. As for games, the successes are numerous. Machines have surpassed human performance in several games, such as backgammon, checkers, chess, and Go (Silver, Schrittwieser, Simonyan, Antonoglou, Huang, Guez, Hubert, Baker, Lai, Bolton, et al., 2017b). A major class of games is the set of two-player games in which players play in turn, without any chance or hidden information. This class is sometimes referred to as two-player *perfect information*[1] games (Mycielski, 1992) or also two-player combinatorial games. Significant challenges remain for this class of games. For example, in the game of Hex, computers have only been able to defeat strong human players since 2020 (Cazenave, Chen, Chen, Chen, Chiu, Dehos, Elsa, Gong, Hu, Khalidov, et al., 2020). In the context of *general game playing* (Genesereth, Love, and Pell, 2005), that is, playing an unknown game with only limited time to learn an effective strategy, humans still outperform machines, even when restricted to perfect-information games. In this article, we focus on two-player zero-sum games with perfect information. The proposed algorithms also apply to the single-player setting. This is demonstrated in the context of Morpion Solitaire.

The first approaches used to design a game-playing program are based on a game tree search algorithm, such as *minimax*, combined with a handcrafted game state evaluation function based on expert knowledge. A notable example of minimax combined with a handcrafted evaluation function is the Deep Blue chess program (Campbell, Hoane Jr, and Hsu, 2002). However, the success of Deep Blue is largely attributable to the computational power available, which enabled the analysis of up to two hundred million game states per second. Moreover, this approach is limited by the need to design an evaluation function manually (at least in part). Designing evaluation functions is a complex task that must be performed separately for each game. Several works have thus focused on automatic learning of evaluation functions (Mandziuk, 2010). One of the first successes in learning evaluation functions was achieved in the game of Backgammon (Tesauro, 1995). However, for many games, such as Hex or Go, minimax-based approaches, with or without machine learning, have failed to surpass human performance. Two causes have been identified (Baier and Winands, 2018). First, the large number of possible actions at each game state prevents exhaustive search to significant depth, limiting lookahead to only a few moves. Second, no sufficiently accurate evaluation function could be identified for these games. An alternative approach addressing these two challenges is Monte Carlo tree search (MCTS) (Coulom, 2007; Browne, Powley, Whitehouse, Lucas, Cowling, Rohlfshagen, Tavener, Perez, Samothrakis, and Colton, 2012). MCTS explores the game tree non-uniformly, which is a solution to the problem of the very large number of actions. In addition, it evaluates game states based on the outcomes of a large number of random end-game simulations. As a result, it does not require a handcrafted evaluation function. MCTS has achieved notably strong results in Hex and Go. However, this was not sufficient to surpass human players. Several variants of Monte Carlo tree search were subsequently proposed, incorporating domain knowledge to guide tree exploration and/or simulation (Browne et al., 2012). Recent improvements in Monte Carlo tree search have focused on the automatic learning and utilization of domain knowledge. This knowledge was initially learned via *supervised learning* (Clark and Storkey, 2015; Gao, Hayward, and Müller, 2017; Gao, Müller, and Hayward, 2018; Cazenave, 2018; Tian and Zhu, 2015) then via a combination of supervised and *reinforcement learning* (Silver, Huang, Maddison, Guez, Sifre,

1. Under some definitions, perfect-information games may include games with chance. This is not the case in this article

Van Den Driessche, Schrittwieser, Antonoglou, Panneershelvam, Lanctot, et al., 2016), and finally using reinforcement learning alone (Silver et al., 2017b; Anthony, Tian, and Barber, 2017; Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2018). These learning-based improvements enabled programs to reach and surpass world-champion-level performance in Go (Silver et al., 2016, 2017b). In particular, AlphaGo Zero (Silver et al., 2017b), which relies solely on reinforcement learning, does not require prior domain knowledge to achieve its level of performance. However, this achievement required approximately 29 million self-play games (with 1,600 state evaluations per move). The AlphaGo Zero approach was subsequently generalized and applied to chess under the name AlphaZero (Silver, Hubert, Schrittwieser, Antonoglou, Lai, Guez, Lanctot, Sifre, Kumaran, Graepel, et al., 2017a). The resulting program outperformed Stockfish, the strongest minimax-based chess engine. Another state-of-the-art zero-knowledge reinforcement learning method is ExIt (Anthony et al., 2017), which is also based on Monte Carlo tree search.

It is therefore unclear whether minimax is obsolete or whether the recent successes of game-playing programs are primarily attributable to reinforcement learning rather than Monte Carlo tree search. It is therefore of interest to examine whether reinforcement learning can provide minimax search with the necessary enhancements to make it competitive with Monte Carlo tree search on games where the latter currently dominates, such as Go or Hex.

In this article[2], we therefore focus on reinforcement learning within the minimax framework. We propose and evaluate new techniques for learning evaluation functions via reinforcement learning, which are then combined to form the Athénan algorithm. In particular, we show that Athénan surpasses the state of the art in reinforcement learning without prior domain knowledge, including the ExIt algorithm, as well as the best specialized methods for the games Hex, Arimaa, Othello, and Morpion Solitaire. In subsequent work conducted in collaboration with Tristan Cazenave, we showed that Athénan is more efficient than AlphaZero (Cohen-Solal and Cazenave, 2023c). This subsequent study complements the results presented in this article.

In the next section, we briefly present game algorithms, in particular Unbounded Best-First Minimax, on which several of our experiments are based. We also introduce reinforcement learning in games and the benchmark games used in our experiments. In the following sections, we propose techniques aimed at improving learning performance and present the corresponding experimental results. In particular, in Section 3, we extend the *tree bootstrapping* (*tree learning*) technique to reinforcement learning without prior knowledge using non-linear function approximation. In Section 4, we present a new search algorithm, a variant of Unbounded Best-First Minimax called *Descent Minimax*, or more succinctly *Descent*, designed for use during the learning process. In Section 6, we introduce *reinforcement heuristics*, which provide a simple mechanism for incorporating general or domain-specific knowledge into reinforcement learning. We study several reinforcement heuristics in the context of different games. In Section 5, we introduce the *completion technique* to account for state resolution in Unbounded Minimax and Descent. Section 7 introduces a new action selection distribution, referred to as the ordinal distribution.

Thereafter, we combine the previously introduced techniques into an algorithm called Athénan and apply it across several settings. In Section 8, we compare Athénan with ExIt (Anthony et al., 2017), a state-of-the-art reinforcement learning algorithm without prior domain knowledge for Hex. In Section 9, we apply Athénan to develop Hex-playing programs (board sizes 11 and 13) and

2. Note that this paper is an extended, improved, and english version of (Cohen-Solal, 2019).

compare them with MoHex 3HNN (Gao et al., 2018), the leading Hex program prior to Athénan, which won the 2018 Computer Olympiad in both sizes (Gao, Takada, and Hayward, 2019). In Section 10, we apply Athénan to surpass the existing state of the art in Othello. In Section 11, we apply Athénan to surpass the existing state of the art in Arimaa. In Section 12, we apply Athénan to match the state-of-the-art performance in Morpion Solitaire.

Moreover, in Section 13, we present Athénan's results at the Computer Olympiad, the world-wide artificial intelligence competition on board games.

Finally, in Section 14, we conclude and discuss directions for future research.

2. Background and Related Work

In this section, we briefly review game tree search algorithms, reinforcement learning in the context of games, and the benchmark games considered in this paper. For further reading, Yannakakis and Togelius (2018) provides a more comprehensive introduction to game algorithms.

Games can be modeled by a *game tree*, where each node represents a game state and each edge corresponds to an action leading to a successor state. Based on this representation, a game tree search algorithm can be used to determine the action to play. Each player aims to maximize their score, defined as the value of the terminal state reached at the end of the game. In the setting of two-player zero-sum games, maximizing one player's score is equivalent to minimizing that of the opponent, as the players' scores are exact opposites.

2.1 Game Tree Search Algorithms

The core algorithm is *minimax* which recursively determines the value of a node from the value of its children and the functions $\min$ and $\max$, up to a specified recursion depth. Standard minimax explores the game tree uniformly. A more efficient implementation employs *alpha-beta pruning* (Knuth and Moore, 1975; Yannakakis and Togelius, 2018) which makes it possible not to explore the sections of the game tree which are less interesting given the values of the nodes already met and the properties of $\min$ and $\max$. Numerous variants and enhancements of minimax have been proposed (Millington and Funge, 2009). For example, *iterative deepening* (Slate and Atkin, 1983; Korf, 1985) allows one to use minimax with a time limit. It performs alpha-beta searches of increasing depth sequentially until the allotted time expires. This approach is typically combined with *move ordering* (Fink, 1982), which extends the most promising move from the previous search first, thereby accelerating subsequent searches.

Some variants perform a search with unbounded depth, meaning that the depth of the search is not fixed (Van Den Herik and Winands, 2008; Schaeffer, 1990; Berliner, 1981). Unlike minimax with or without alpha-beta pruning, these algorithms explore the game tree non-uniformly. One example is *best-first minimax search* (Korf and Chickering, 1996)[3]. To avoid confusion with other best-first approaches that operate at a fixed depth, we refer to this algorithm as Unbounded Best-First Minimax, or UBFM for short.

UBFM iteratively expands the game tree by adding the children of one of the leaves whose minimax value matches that of the root. These leaves correspond to states reached by following one of the best action sequences given the current partial knowledge of the tree. Consequently,

3. During the initial experiments reported in this article, we were not aware that the Unbounded Best-First Minimax algorithm had previously been proposed, and thus we independently rediscovered it.

the algorithm progressively extends the a priori most promising sequences of actions. As the search proceeds, the set of best sequences typically changes at each expansion. In this way, the game tree is explored non-uniformly, concentrating on the most promising actions while avoiding the restriction to a single sequence of moves.

In this article, we use the anytime version of UBFM (Korf and Chickering, 1996), in which a fixed search time is allocated for UBFM to select an action. We also use transposition tables (Greenblatt, Eastlake, and Crocker, 1988; Millington and Funge, 2009) with UBFM, which allows the algorithm to avoid building the entire game tree by merging nodes corresponding to identical states (MCTS and Alpha-Beta search algorithms considered in this paper also employ transposition tables). The positive impact of transposition tables on UBFM has been evaluated (Cohen-Solal and Cazenave, 2025b). Algorithm 1 is an implementation of UBFM[4]. UBFM has been modified into the algorithm called *Unbounded Minimax with Safe Decision* (Cohen-Solal, 2025), denoted $\mathrm{UBFM_s}$. This variant conducts the same search of the game tree but selects a different action based on the search results. Rather than playing the action leading to the state with the highest value, $\mathrm{UBFM_s}$ chooses the action that was most frequently selected from the current state during the search. This variant of Unbounded Minimax was initially proposed in earlier, unpublished versions of this article (Cohen-Solal, 2020), but has been omitted here for clarity and to reduce content. The results concerning Unbounded Minimax with Safe Decision, which were omitted from the previous versions, have been refined and presented in detail in a dedicated article introducing the algorithm (Cohen-Solal, 2025).

Remark 1. In the experiments presented in this paper, the transposition tables are only cleared at the end of the game. Thus, states explored during one turn are reused in the next.

2.2 Learning of Evaluation Functions

Reinforcement learning of evaluation functions can be done by different techniques (Mandziuk, 2010; Silver et al., 2017b; Anthony et al., 2017; Young, Vasan, and Hayward, 2016). The general idea of reinforcement learning of state evaluation functions is to use a game tree search algorithm and an adaptive evaluation function to play a sequence of matches (for example against oneself, which is the case in this article). Adaptive evaluation functions f_θ are usually neural networks, parameterized by θ. Each match generates pairs (s, v) where s is a game state and v is the value of s calculated by the chosen search algorithm using the evaluation function f_θ. Such pairs are learned for updating f_θ in order to improve it. The set of states employed to update f_θ depends on the specific learning technique (Tesauro, 1995; Veness, Silver, Blair, and Uther, 2009). For example, in the case of *root bootstrapping* (technique that we call *root learning* in this article), the set of pairs learned during the learning phase is $D = \{(s, v) \mid s \in \mathbf{R}\}$ with $\mathbf{R}$ the set of states of the match. In the case of the *tree bootstrapping* (*tree learning*) technique (Veness et al., 2009), the set of pairs learned during the learning phase is the set of states of the partial game tree built to decide which actions to play: $D = \{(s, v) \mid s \in \mathbf{T}\}$ with $\mathbf{T}$ the set of states of the search trees. Importantly, the states in the match search trees include the match states, as these states form the roots of the search trees. Thus, contrary to root bootstrapping, tree bootstrapping does not discard most of the

4. This implementation is a slight variant of Korf and Chickering algorithm. Both algorithms behave identically if the evaluation function assigns a unique value to each state and transposition tables are not used. In practice, when using transposition tables, our variant improves the winning rate by several percentage points (Cohen-Solal and Cazenave, 2025b). If the evaluation function assigns a unique value to each state and transposition tables are not used, Korf and Chickering's algorithm is marginally faster.

information used to decide the actions to play. The values of the generated states can be their minimax values in the partial game tree built to decide which actions to play (Veness et al., 2009; Tesauro, 1995). Work on tree bootstrapping has been limited to reinforcement learning of linear functions of state features. It has not been formulated or studied in the context of reinforcement learning without knowledge and based on non-linear functions. Note that, in the case of AlphaGo Zero, the learned states are the states of the sequence of the match (as with root learning), but the target value for each of these states is the value of the terminal state of the match (Silver et al., 2017b). We call that technique terminal learning.

Typically, a *learning phase* occurs between two consecutive matches, using the state–value pairs generated in the previous match. Each learning phase modifies f_θ so that, for all pairs $(s, v) \in D$, $f_\theta(s)$ closely approximates v. However, learning phases can also use the pairs of *several* matches. This technique is called *experience replay* (Mnih, Kavukcuoglu, Silver, Rusu, Veness, Bellemare, Graves, Riedmiller, Fidjeland, Ostrovski, et al., 2015). Remark that, adaptive evaluation functions f_θ only serve to evaluate non-terminal states since we know the true value of terminal states. More precisely, if the exact terminal evaluation function is not very costly (relative to f_θ), it is preferable to use it, as it has been shown to improve performance experimentally (Cohen-Solal and Cazenave, 2025b).

Symbols	Definition
$\mathrm{actions}\,(s)$	action set of the state s for the current player
$\mathrm{first\_player}\,(s)$	true if the current player of the state s is the first player
$\mathrm{terminal}\,(\mathrm{s})$	true if s is an end-game state
$a(s)$	state obtained after playing the action a in the state s
$\mathrm{time}\,()$	current time in seconds
$\mathrm{random}\,()$	returns a uniformly random number from $[0, 1]$
S	keys of the transposition table T
T	transposition table (contains states labels as function v ; depends on the algorithm)
τ	search time per action
t	time elapsed since the start of the reinforcement learning process
$t_{\max}$	chosen total duration of the learning process
$n(s, a)$	number of times the action a is selected in state s (initially, $n(s, a) = 0$ for all s and a
$v(s)$	value of state s in the game tree
$v(s, a)$	value of the state reached after playing action a in state s
$c(s)$	completion value of state s (0 by default)
$c(s, a)$	completion value of the state reached after playing action a in state s
$r(s)$	resolution value of state s (0 by default)
$r(s, a)$	resolution value of the state reached after playing action a in state s
$f(s)$	the used evaluation function (first player point of view)
$f_\theta(s)$	adaptive evaluation function (of non-terminal game tree leaves ; first player view)
$f_{\mathrm{t}}(s)$	evaluation of terminal states, e.g. gain game (first player point of view)
Gain function $\mathrm{b_t}(s)$	0 if s is a draw, 1 if s is a win for the first player, -1 if s is a loss for the first player
search(s, S, T, f_θ, f_{t})	a seach algorithm (it extends the game tree from s, by adding new states in S and labeling its states, in particular, by a value $v(s)$, stored in T, using f_θ as evaluation of the non-terminal leaves and f_{t} as evaluation of terminal states
action_selection(s, S, T)	decides the action to play in s depending on the partial game tree, i.e. on S and T
processing(D)	various optional data processing: data augmentation (symmetry), experience replay, ...
update(f_θ, D)	updates the parameter θ of f_θ in order for $f_\theta(s)$ is closer to v for each $(s, v) \in D$

Table 1: Index of symbols (s is a game state ; n, v, c, r are values tables, i.e. dictionary).

Remark 2. There is another reinforcement learning technique for games: the Temporal Differences algorithm $\mathrm{TD}(\lambda)$ (Tesauro, 1995). It corresponds to a root learning algorithm without search (i.e. with a search of depth 1) where the target of a state is a weighted average of the values of its successor states in the match, parameterized by the constant λ. A variant of $\mathrm{TD}(\lambda)$ has also been proposed (Baxter, Tridgell, and Weaver, 2000), called $\mathrm{TD}_{\mathrm{Leaf}}(\lambda)$, where the temporal difference technique $\mathrm{TD}(\lambda)$ is modified by applying it with a minimax search (i.e. with a search of depth $d \geq 1$). A comparison between these techniques was made (Veness et al., 2009).

Machine learning has also focused on how to explore game trees: a method has been described for automatically tuning *search-extension* parameters, to decide which branches of the game tree must be explored during the search (Björnsson and Marsland, 2003) .

2.3 Action Selection Distribution

One of the problems related to reinforcement learning is the *exploration-exploitation dilemma* (Mandziuk, 2010). This dilemma involves choosing between exploring new states to acquire knowledge and exploiting previously acquired knowledge. Many techniques have been proposed to deal with this dilemma (Mellor, 2014). However, most of these techniques do not scale because their application requires memorizing all the encountered states. For this reason, in the context of games

Algorithm 1 The UBFM (Unbounded Best-First Minimax) algorithm. It computes the best action to play in the generated partial game tree (see Table 1 for symbol definitions). At any time, $T = \{v(s,a) \,|\, s \in S, a \in \mathrm{actions}(s)\}$, where $v(s,a)$ denotes the last stored minimax value estimate of the state $a(s)$, i.e., its minimax value in the current partial game tree computed after visiting s).

```
Function UBFM_iteration(s, S, T)
    if terminal(s) then
        return f(s)
    else
        if s ∉ S then
            S ← S ∪ {s}
            foreach a ∈ actions(s) do
                v(s, a) ← f(a(s))
        else
            a_b ← best_action(s)
            v(s, a_b) ← UBFM_iteration(a_b(s), S, T)
        a_b ← best_action(s)
        return v(s, a_b)
Function best_action(s, S, T)
    if first_player(s) then
        return arg max_{a∈actions(s)} v(s, a)
    else
        return arg min_{a∈actions(s)} v(s, a)
Function UBFM(s, τ)
    t = time() while time() − t < τ do UBFM_iteration(s, S, T)
    return best_action(s, S, T)
```

with large numbers of states, some approaches use probabilistic exploration (Young et al., 2016; Silver et al., 2017b; Mandziuk, 2010; Schraudolph, Dayan, and Sejnowski, 2001). Under this approach, exploitation corresponds to selecting the best action, while exploration corresponds to selecting an action uniformly at random. More precisely, a parametric probability distribution is used to associate with each action its probability of being played. The parameter of the distribution defines the exploration rate $\epsilon \in [0, 1]$; the corresponding exploitation rate is $1 - \epsilon$, which we denote ϵ'. The rate is often experimentally fixed. *Simulated annealing* (Kirkpatrick, Gelatt, and Vecchi, 1983) can, however, be applied to avoid choosing a value for the parameter. In this case, the parameter is initially set to 1, corresponding to pure exploration, and it gradually decreases to 0 by the end of the learning process. The simplest action selection distribution is *ϵ-greedy* (Young et al., 2016) (of parameter ϵ). With the distribution *ϵ-greedy*, the action is chosen uniformly with probability ϵ and the best action is chosen with probability $1 - \epsilon$ (see also Algorithm 2).

Algorithm 2 The ϵ-greedy algorithm with simulated annealing ($\epsilon = 1 - \frac{t}{t_{\max}}$) which was used in the experiments reported in this article (see Table 1 for symbol definitions).

Function ϵ_greedy(*s*, *v*)
  if random() $\leq \frac{t}{t_{\max}}$ **then**
    if first_player(*s*) **then**
      return $\arg\max_{a \in \texttt{actions}(s)} v(s, a)$
    else
      return $\arg\min_{a \in \texttt{actions}(s)} v(s, a)$
  else
    return $a \in$ actions(*s*) uniformly chosen.

A limitation of the ϵ-greedy distribution is that it does not differentiate the probabilities of actions, except for the best action. Another distribution is often used, correcting this disadvantage. This is the *softmax* distribution (Schraudolph et al., 2001; Mandziuk, 2010). It is defined by $P(a_i) = \frac{e^{v(s,a_i)/\tau}}{\sum_{j=1}^{n} e^{v(s,a_j)/\tau}}$ with n the number of children of the current state s, $P(a_i)$ the probability of playing the action a_i , $v(s, a_i)$ the value of the state obtained after playing a_i in the current state s, $i \in \{0, \ldots, n-1\}$, and $\tau \in]0, +\infty[$ a parameter called temperature ($\tau \simeq 0$: exploitation, $\tau \simeq +\infty$: exploration).

2.4 Games of Experiments

We now briefly present the games on which experiments are performed in this article, namely: Hex, Arimaa, Morpion Solitaire, Othello, Surakarta, Outer Open Gomoku, Clobber, Breakthrough, Amazons, Lines of Action, and Santorini. They are all board games. All of these games are present and recurring (except Morpion Solitaire) at the Computer Olympiad, the worldwide multi-games event in which computer programs compete against each other. Moreover, all these games (and their rules) are included (and available for free) in Ludii (Piette, Soemers, Stephenson, Sironi, Winands, and Browne, 2020), a general game system.

2.4.1 HEX

Hex is a two-player combinatorial strategy game played on an $n \times n$ hexagonal board. Each player takes turns placing a stone of their color on an empty cell. The objective is to be the first to connect

the two opposite sides of the board corresponding to their color. A common variant of Hex includes the swap rule, where the second player can choose, on their first turn, to swap roles and sides with the first player. This rule helps balance the game by mitigating the first player's advantage and is typically used in competitions. It is still used at the Computer Olympiad, and we use it in this paper.

2.4.2 ARIMAA

Arimaa is a game similar to Chess. Each player has a set of pieces: one elephant, one camel, two horses, two dogs, two cats, and eight rabbits. The pieces types are fully ordered: elephant $>$ camel $>$ horse $>$ dog $>$ cat $>$ rabbit. A higher piece can move (push or pull) a lower piece. A piece which is orthogonally adjacent to a stronger opposing piece is frozen, unless it is also adjacent to a friendly piece. If a piece is on a trap square without being adjacent to a friendly piece, it is removed from the board. A player wins when one of its rabbits reaches the other side of the board. A player without rabbits loses. It is played on a 8×8 grid like Chess (the cases $(2, 2)$, $(2, 5)$, $(5, 2)$, $(5, 5)$ are traps). At the start, the board is empty. The first player places its pieces on its side of the board then the second player does the same. On their turn, a player can make up to four single-square moves, possibly using multiple pieces. Pushing or pulling an opposing piece counts as two moves. Details about rules are available in `https://arimaa.com/arimaa/`.

2.4.3 MORPION SOLITAIRE

The game is played on a grid of unlimited size. The starting configuration includes a set of points already placed on the grid in the shape of a Greek cross whose side has 4 points (the cross is made up of a total of 36 points).

A move in this game consists of two steps. First, the player places a new point on the grid so as to form an alignment of five adjacent points horizontally, vertically or diagonally (which have not already been connected by a line). Then, he connects this alignment by drawing a line. An alignment cannot be placed as an extension of a previous alignment.

The aim of the game is to place as many points as possible before reaching a situation where no new points can be placed.

2.4.4 OTHELLO

Othello (also called Reversi) is a territory and linear encirclement game whose goal is to have more pieces than its opponent. In his turn, a player places a piece of his color on the board (only if he can make an encirclement, otherwise he pass his turn). There is an encirclement if an opponent's line of pieces has at one of its ends the piece that the current player has just placed, and has at the other end, another piece of the current player. As a result of this encirclement, the encircled opponent's pieces are replaced by pieces of the current player.

2.4.5 SURAKARTA

Surakarta is a move and capture game (like draughts). The goal of the game is to take all the opposing pieces. In his turn, a player can either move a piece to an empty square at a distance of 1 or move a piece to a square occupied by an opponent's piece under certain conditions and according to a mechanism specific to Surakarta (based on a movement circuit dedicated only to capture), allowing "long distance" capture. Any repetition or 50 turns without capture (25 player

turns for each player) triggers the end of the game. The winner is then the one with the most pieces (in case of a tie, it is a draw).

2.4.6 OUTER OPEN GOMOKU

Outer Open Gomoku is an alignment game. The goal of the game is to line up at least 5 pieces of its color. On their turn, a player places a piece of their color. On the first turn, the first player must place a piece at a distance of two squares from the board edges.

2.4.7 CLOBBER

Clobber is a move and capture game. The goal is to be the last player to have played. A player can play if he can orthogonally move one of his pieces onto a neighboring square on which there is an opponent's piece. This movement is always a capture (the opponent's piece is removed from the board).

2.4.8 BREAKTHROUGH

Breakthrough is a move and capture game. The goal of the game is to be the first to make one of his pieces reach the other side of the board. A piece can only move by moving forward one square (straight or diagonal). A capture can only be done diagonally.

2.4.9 AMAZONS

Amazons is a move and blocking game. In turn, a player moves one of his pieces in a straight line in any direction (like the queen of the game of Chess). Then he places a neutral piece in any direction starting from the new position of the piece just moved (always in the manner of the queen of the game of Chess). Any piece, whether neutral or player-owned, blocks both placement and movement. The goal of the game is to be the last to play.

2.4.10 LINES OF ACTION

Lines of Action is a game of movement and regrouping. On his turn, a player can move one of his pieces in one direction as many squares as there are pieces in that direction. A piece cannot move if there is an opponent's piece in its path, unless it is the square to arrive (in which case a capture is made). The goal is to have all of its pieces connected. We modified the repetition rule to avoid overly long matches: the first repetition triggers the end of the game.

2.4.11 SANTORINI

Santorini is a three-dimensional building and moving game. The goal of the game is to reach the 3rd floor of a building. In his turn, a player moves one of his pieces by one square then places the first floor on an adjacent empty square or increases a pre-existing construction by one floor (on which no player's piece is located). A piece cannot move to a square that is two or more levels higher than its current square (a piece can move up by only one floor at a time and may descend any number of floors). A move cannot be made to a square with 4 floors. A construction cannot be done on a square of 4 floors. A player who cannot play loses. The advanced mode (i.e. the use of power cards) is not used in the experiments in this article.

3. Data Use in Game Learning

In this section, we adapt and study tree learning (see Section 2.2) in the context of reinforcement learning and the use of non-linear adaptive evaluation functions. For this purpose, we compare it to root learning and terminal learning. We start by adapting tree learning, root learning, and terminal learning. We also define in Section 3.2 a variant of experience replay that we call stratified experience replay. Next, we describe the experiment protocol common to several sections of this article. Finally, we expose the comparison of tree learning with root learning and terminal learning.

3.1 Tree Learning

As we saw in Section 2.2, tree learning consists of learning the values of the states in the search tree. However, our tree learning technique has two main differences with classic tree learning (in addition to the generalization to reinforcement learning without knowledge and with non-linear functions). On the one hand, the learning phase is carried out only after each game (and not after each search, i.e. not after each action play). On the other hand, the data of the non-terminal leaves of the tree are not learned. Thus learning is performed on the partial game tree obtained at the end of the match from which the non-terminal leaves have been removed. This approach drastically reduces the amount of data to be learned and could prevent overfitting, thereby significantly improving learning performance.

As a reminder, root learning consists in learning the values of the states of the states sequence of the match (the training target value of each state is its value in the search tree). Terminal learning consists also in learning the values of the states sequence of the match but the target value of each state is the value of the terminal state of the match (i.e. the gain of the match). To generalize tree learning and root learning, in a manner similar to the use of terminal learning in the AlphaZero framework, data to learn after each game can be modified by some optional data processing methods, such as experience replay (which involves learning a sample of previous matches instead of just learning the entire data from the previous match). Additionally, the learning phase employs a specific update method to ensure that the adaptive evaluation function fits the selected data. The adaptation of tree learning, root learning, and terminal learning are given respectively in Algorithm 3, Algorithm 4, and Algorithm 5. The data processing method Experience replay is described in Algorithm 6 (its parameter are the memory size μ and the sampling rate σ). In addition, we use in this article a stochastic gradient descent as update method (see Algorithm 7 ; its parameter is B the batch size). Formally, in Algorithm 3, Algorithm 4, and Algorithm 5, we have: processing(D) is experience_replay(D, μ, σ) and update(f_θ, D) is stochastic_gradient_descent(f_θ, D, B). Finally, we use ϵ-greedy as default action selection method, i.e. action_selection(s, S, T) is ϵ-greedy(s, $T.v$) (see Algorithm 2 ; recall that T stores the children value table $v(s, a)n$).

Remark 3. The exclusion of non-terminal leaves is motivated by considerations of both computational efficiency and learning dynamics. In most games, the vast majority of states in the partial game tree are non-terminal leaves (e.g., approximately 95% for branching factor 100 ; of the order of $\frac{b^d \cdot (b-1)}{b^{d+1}-1}$ for branching factor b and tree depth d). Since their values are evaluated using the same network being trained, including them in the target set largely amounts to training the model on its own current predictions, which is both redundant and potentially harmful.

In particular, this self-reinforcement could amplify approximation errors, dilute more informative signals (e.g., terminal leaves or internal nodes), and increase the risk of overfitting. It also significantly increases training cost (by approximately a factor of 100 for branching factor 100 ; by

approximately a factor of $\frac{b^{d+1}-1}{b^d-1}$ for branching factor b and tree depth d), as non-terminal leaves dominate the target distribution. Empirically, we observed degraded performance and memory issues when including them. While we do not provide a dedicated ablation isolating this choice, our observations, not presented in this article, support improved efficiency when excluding non-terminal leaves. However, for games with smaller branching factors and appropriately tuned hyperparameters, including non-terminal leaves could be advantageous.

Algorithm 3 Tree learning (tree bootstrapping) algorithm (see Table 1 for the definitions of symbols). For tree learning, S is the set of states which are non-leaves or terminal.

Function `tree_learning` ($t_{\max}$, τ)
- $t_0 \leftarrow$ `time()`
- **while** `time()` $- t_0 < t_{\max}$ **do**
  - $s \leftarrow$ `initial_game_state()`
  - $S \leftarrow \emptyset$
  - $T \leftarrow \{\}$
  - **while** $\neg\text{terminal}(s)$ **do**
    - S, $T \leftarrow$ `search` (s, S, T, f_θ, f_{t}, τ)
    - $a \leftarrow \text{action\_selection}(s, S, T)$
    - $s \leftarrow a(s)$
  - $D \leftarrow \{(s, v(s)) \mid s \in S\}$
  - $D \leftarrow$ `processing` (D)
  - `update` (f_θ, D)

Algorithm 4 Root learning (root bootstrapping) algorithm (see Table 1 for the definitions of symbols).

Function `root_learning` ($t_{\max}$, τ)
- $t_0 \leftarrow$ `time()`
- **while** `time()` $- t_0 < t_{\max}$ **do**
  - $s \leftarrow$ `initial_game_state()`
  - $S \leftarrow \emptyset$
  - $T \leftarrow \{\}$
  - $D \leftarrow \emptyset$
  - **while** $\neg\text{terminal}(s)$ **do**
    - S, $T \leftarrow$ `search` (s, S, T, f_θ, f_{t}, τ)
    - $a \leftarrow \text{action\_selection}(s, S, T)$
    - $D \leftarrow D \cup \{(s, v(s))\}$
    - $s \leftarrow a(s)$
  - $D \leftarrow D \cup \{(s, v(s))\}$
  - $D \leftarrow$ `processing` (D)
  - `update` (f_θ, D)

3.2 Stratified Experience Replay

In this section, we introduce the technique we call *Stratified Experience Replay*, which is a variation of Experience Replay. The targeted objective of Stratified Experience Replay is to reduce the variance of learning processes.

Algorithm 5 Terminal learning algorithm (see Table 1 for the definitions of symbols).

Function `terminal_learning` ($t_{\max}$, τ)
- $t_0 \leftarrow$ `time()`
- **while** `time()` $- t_0 < t_{\max}$ **do**
  - $s \leftarrow$ `initial_game_state()`
  - $S \leftarrow \emptyset$
  - $T \leftarrow \{\}$
  - $G \leftarrow \{s\}$
  - **while** $\neg$terminal (s) **do**
    - S, $T \leftarrow$ `search` (s, S, T, f_θ, f_{t}, τ)
    - $a \leftarrow$ action_selection (s, S, T)
    - $s \leftarrow a(s)$
    - $G \leftarrow G \cup \{s\}$
  - $D \leftarrow \{(s', f_{\mathrm{t}}(s)) \mid s' \in G\}$
  - $D \leftarrow$ `processing` (D)
  - `update` (f_θ, D)

Algorithm 6 Experience replay (replay buffer) algorithm. Its parameters are μ the memory size and σ the sampling rate. M is the memory buffer: a global variable initialized by an empty queue. If the number of data is less than $\sigma \cdot \mu$, then the algorithm returns all data (no sampling). Otherwise, it returns $\sigma \cdot \mu$ random elements.

Function `experience_replay` (D, μ, σ)
- elements of D are added in M
- **if** $|M| > \mu$ **then**
  - remove the oldest items of M to have $|M| = \mu$
- **if** $|M| \leq \sigma \cdot \mu$ **then**
  - return M
- return a list of random items of M whose size is $\sigma \cdot \mu$

Algorithm 7 Stochastic gradient descent algorithm used in the experiments of this article. It is based on Adam optimization (1 epoch per update) (Kingma and Ba, 2014) with L_2 regularization ($\lambda = 0.001$) (Ng, 2004) and implemented in tensorflow. B denotes the batch size (see Table 1 for the definitions of other symbols).

Function `stochastic_gradient_descent` (f_θ, D, B)
- Split D in m disjoint sets, denoted $\{D_i\}_{i=1}^m$, such that m is minimal, $D = \bigcup_{i=1}^m D_i$, $|D_i| = B$ for each $i \in \{1, \ldots, m-1\}$, and $|D_m| \leq B$
- **foreach** $i \in \{1, \ldots, m\}$ **do**
  - minimize $\sum_{(s,v) \in D_i} (f_\theta(s) - v)^2$ by using Adam and L_2 regularization

Stratified Experience Replay consists of performing stratified sampling without replacement on the replay buffer of Experience Replay, where the data are grouped according to the matches that generated them. Stratified sampling imposes the same proportion of data for each previous match, so we have a good level of variability during each learning phase. The absence of replacement ensures that no data are unnecessarily repeated and that no data are omitted.

Stratified Experience Replay depends on two parameters: μ the number of previous matches kept in memory and δ the duplication factor: the number of times a piece of data must be learned during the learning phases. The data is stored in two data sets, the data to be learned stored in μ sets D_i with $i \in \{1, \dots, \mu\}$, each corresponding to the data of the i-th previous match that must be learned and μ other sets D'_i with $i \in \{1, \dots, \mu\}$, the archived data, which correspond to the data already learned from the i-th previous match. The data set for the new learning phase is then formed by sampling from each D_i with sample size $\left\lceil \frac{\delta}{\mu} \left(|D_i| + |D'_i| \right) \right\rceil$, so that the total amount of data to be learned equals the number of generated data multiplied by the duplication factor δ. The selected samples are removed from the D_i and added to the D'_i . The selected data is then separated into minibatches, so as to preserve the stratification as much as possible (the minibatches are thus in a way also stratified) and learning is then performed on each minibatch. The formalization of Stratified Experience Replay is described in Algorithm 8.

Remark 4. We were unable to evaluate the benefits or drawbacks of this variant compared to standard experience replay: in preliminary experiments (not reported here), the observed performance differences were so small that distinguishing them would require increasing the number of repetitions by an order of magnitude, which is currently infeasible. We make no claim that this variant outperforms, or is inferior to, standard experience replay.

Remark 5. On some games, and with certain algorithms and parameter settings, stratified experience replay can occasionally lead to memory overflows. During training, in these rare cases, self-play matches can suddenly become much longer, consequently increasing the amount of memory used. To address this issue in our implementation, the buffer size is temporary dynamically reduced by removing the data of the oldest matches (by removing as few matches as possible from the buffer).

3.3 Common Experiment Protocol

The experiments of several sections share the same protocol. We present it in this section. The protocol is used to compare different variants of reinforcement learning algorithms. A variant corresponds to a specific combination of fundamental algorithmic components. More specifically, a combination consists of the association of a search algorithm (iterative deepening alpha-beta (with move ordering), MCTS (UCT with $c = 0.4$ as exploration constant), UBFM, ...), of an action selection method (ϵ-greedy distribution (used by default), softmax distribution, ...), a terminal evaluation function f_{t} (the classic game gain (used by default), ...), and a procedure for selecting the data to be learned (root learning, tree learning, or terminal learning). The protocol involves running a reinforcement learning process for 48 hours for each variant and for each repetition. At several stages of the learning process, each combination is evaluated. This evaluation consists in matches played by using the adaptive evaluation function generated by the evaluated combination. Each variant is thus characterized by a winning percentage at each stage of the reinforcement learning process. More formally, we denote by $f^c_{\theta_h}$ the evaluation generated by the combination c at the hour h. Each combination is evaluated every hour by a winning percentage. The winning percentage of a combination c at hour $h \leq 48$ is computed from matches using a depth-1 minimax search with $f^c_{\theta_h}$

as the evaluation function, against opponents consisting of depth-1 minimax searches with $f^{c'}_{\theta_{48}}$ as evaluation functions for all studied combinations c'. For each pair of combinations, one match is played with each as first player.

This protocol is repeated several times for each experiment in order to reduce the statistical noise in the winning percentages obtained for each variant (the obtained percentage is the average of the percentages of repetitions). The winning percentages are then represented in a graph showing the evolution of the winning percentages during training.

In addition to the curve, the different variants are also compared in relation to their final winning percentage, i.e. at the end of the learning process. Unlike the evolution of winning percentages, in the comparison of final performances, each evaluation $f^{c}_{\theta_{48}}$ confronts each other evaluation $f^{c'}_{\theta_{48}}$ from all repetitions. In other words, this experiment consists in performing an all-play-all tournament

Algorithm 8 Stochastic gradient descent with Stratified experience replay algorithm. Its parameters are B the batch size, μ the memory size, and δ the duplication factor. M is the memory buffer: a global variable initialized by an empty queue. Method choice(E) uniformly randomly chooses an element of E. See Table 1 for definitions of the other symbols.

Function `SGD_with_stratified_experience_replay(`f_θ, D, μ, δ, B`)`
- append $(D, \emptyset)$ in M
- **if** $|M| > \mu$ **then**
  - remove the oldest items of M to have $|M| = \mu$
- $n \leftarrow \sum \left\{ \left\lceil \frac{\delta}{\mu} (|D| + |D'|) \right\rceil \mid (D, D') \in M \right\}$
- $g \leftarrow \max(1, \lfloor n/B \rfloor)$
- **if** $|\lfloor n/g \rfloor - B| > |\lfloor n/(g+1) \rfloor - B|$ **then**
  - $g \leftarrow g + 1$
- $G \leftarrow \{1, \ldots, g\}$
- **foreach** $i \in G$ **do**
  - $S_i \leftarrow \emptyset$
- **foreach** $(D, D') \in M$ **do**
  - $l \leftarrow \left\lceil \frac{\delta}{\mu} (|D| + |D'|) \right\rceil$
  - **foreach** $c \in \{1, \ldots, l\}$ **do**
    - $(s, v) \leftarrow \text{choice}(D)$
    - remove (s, v) from D
    - add (s, v) in D'
    - **if** $|D| = 0$ **then**
      - $D \leftarrow D'$
      - $D' \leftarrow \emptyset$
    - $j \leftarrow \text{choice}(G)$
    - $G \leftarrow G \backslash \{j\}$
    - **if** $|G| = 0$ **then**
      - $G \leftarrow \{1, \ldots, g\}$
    - add (s, v) in S_j
- **foreach** $i \in \{1, \ldots, g\}$ **do**
  - minimize $\sum_{(s,v) \in S_i} (f_\theta(s) - v)^2$

with all the evaluation functions generated during the different repetitions. The presented winning percentage of a combination is still the average over the repetitions. The matches are also made by using minimax at depth 1. These percentages are shown in tables.

Remark 6. The version of MCTS used does not perform random simulations to evaluate the leaves. Instead, leaves are evaluated by a neural network. No policies are used (unless it is explicitly specified).

Remark 7. All the experiments based on this protocol involving MCTS were also performed with $c = \sqrt{2}$ as exploration constant. Using the alternative value $c = \sqrt{2}$ yields the same conclusions, in particular that Descent with tree learning is the most effective combination across all games.

Remark 8. Optimizer schedules are constant in all experiment of this paper.

Remark 9. A limitation of the common experimental protocol is that it includes a partially circular component, where some models are evaluated against themselves or later versions of themselves. In the final evaluation, all models are fully trained, and this circular component accounts for only a small fraction of matches $1/192$, so its impact on the reported results is minimal.

For the learning curves, evaluations involve models from different training stages, so the comparisons are not strictly circular. Although a small bias cannot be entirely excluded, its effect is likely limited. Furthermore, the performance differences between methods are sufficiently large that this limitation is unlikely to affect the overall conclusions. A fully non-circular evaluation protocol would be preferable, but would require rerunning the entire experimental pipeline. Moreover, this limitation only affects the ablation studies of the proposed techniques and does not apply to the experiments comparing with state-of-the-art algorithms. More specifically, this protocol is not used for the primary experiments in this paper, which show that Athénan reaches or exceeds ExIt and the dedicated state of the art in Hex, Othello, Arimaa, and Morpion Solitaire.

3.3.1 TECHNICAL DETAILS

We now present the technical details of this protocol.

The training time for each neural network is 48 hours. The parameters used are: search time per action $\tau = 2s$, batch size $B = 128$, memory size $\mu = 10^6$, sampling rate $\sigma = 4\%$ (see Section 3.1). Moreover, the adaptive evaluation function for each combination is a convolutional neural network (Krizhevsky, Sutskever, and Hinton, 2012) with three convolution layers[5] followed by a fully connected hidden layer. For each convolutional layer, the kernel size is 3×3 and the filter number is 64. The number of neurons in the fully connected layer is 100. The margin of each layer is zero. After each layer except the last one, the ReLU activation function (Glorot, Bordes, and Bengio, 2011) is used. The output layer contains a neuron. When the classical terminal evaluation is used, $\tanh$ is the output activation function. Otherwise, there is no activation function for the output. Adam (Kingma and Ba, 2014) is the optimizer (Its parameters are $\lambda = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$). L_2 regularization is used with 0.001 as parameter.

Hardware used with the common experiment protocol: a quarter of the following node: 2 Intel Cascade Lake 6248 processors (20 cores at 2.5 GHz), 192 GB of memory, with 4 Nvidia Tesla V100 SXM2 32GB GPUs.

5. There is an exception: for the game Surkarta, there is only two convolution layers.

3.4 Comparison of Learning Data Selection Algorithms

We now compare tree learning, root learning and terminal learning, using the protocol of Section 3.3. Each combination uses either tree learning, root learning, or terminal learning. Moreover, each combination employs either iterative deepening alpha-beta (denoted by ID) or MCTS. Furthermore, each combination uses ϵ-greedy as action selection method (see Section 3.1) and the classical terminal evaluation (1 if the first player wins, -1 if the first player loses, and 0 in case of a draw). There are a total of 6 combinations. The experiment was repeated 32 times. The winning percentage of a combination for each game and for each evaluation step (i.e. each hour) is therefore calculated from 384 matches (recall that there is a first player match and a second player match per pair of combinations).

The winning percentage curves are shown in Figure 1. The final winning percentages are shown in Table 2. Each percentage of the table has required $12,288$ matches. ID is first on all games except in Outer Open Gomoku where it is second (MCTS root learning is first) and in Surakarta (MCTS with tree learning is first). MCTS with root learning is better than MCTS with tree learning except in Breakthrough and Surakarta. At Hex and Amazons, MCTS with root learning gives better results throughout the learning process but ends up being caught up by ID with terminal learning. Terminal learning generally performs worse, except in a few cases where it shows marginal improvements. On average, ID with tree learning is better (71% win), then MCTS with root learning is second (9% lower win percentage), followed by MCTS with tree learning (18% lower to ID).

In conclusion, tree learning with ID outperforms the other combinations, although the differences are small for Amazons, Hex, and Outer Open Gomoku when using MCTS with root learning.

	tree learning		root learning		terminal learning	
	MCTS	ID	MCTS	ID	MCTS	ID
Othello	48.8%	55.4%	51.0%	31.9%	52.9%	43.7%
Hex	45.1%	79.8%	79.8%	44.1%	29.8%	27.8%
Clobber	41.5%	62.5%	50.0%	45.5%	45.7%	49.8%
Outer Open Gomoku	40.3%	80.0%	87.3%	48.8%	21.6%	23.1%
Amazons	46.2%	58.8%	56.1%	44.5%	39.4%	46.0%
Breakthrough	78.1%	79.2%	45.5%	50.6%	22.3%	14.3%
Santorini	50.2%	73.8%	60.5%	51.3%	29.4%	36.5%
Surakarta	69.4%	65.2%	56.2%	20.8%	28.9%	23.6%
Lines of Action	58.8%	81.7%	67.7%	9.6%	6.9%	2.7%
mean	53.1%	70.7%	61.6%	38.5%	30.8%	29.7%

Table 2: Final winning percentages of the combinations of the experiment of Section 3.4 (ID: iterative deepening alpha-beta). Reminder: the percentage is the average, over repetitions, of the win rate obtained by neural networks generated by the evaluated combination when playing matches against networks generated by all other combinations across all repetitions, considering both first- and second-player roles (see Section 3.3).

4. Tree Search Algorithms for Game Learning

In this section, we introduce a new tree search algorithm, *Descent Minimax* (or simply *Descent*), designed for use during the learning process. After presenting Descent, we compare it to MCTS (with root and tree learning), iterative deepening alpha-beta (with root and tree learning), and UBFM with tree learning.

4.1 Descent: Generate Better Data

Thus, we present *Descent*. It is a modification of UBFM that constructs a deeper game tree and is designed to be combined with tree learning (see Algorithm 3).

Remark 10. Note that, combining it with root learning or terminal learning is of no interest. The reason is that Descent spends most of its time exploring parts of the game tree whose primary

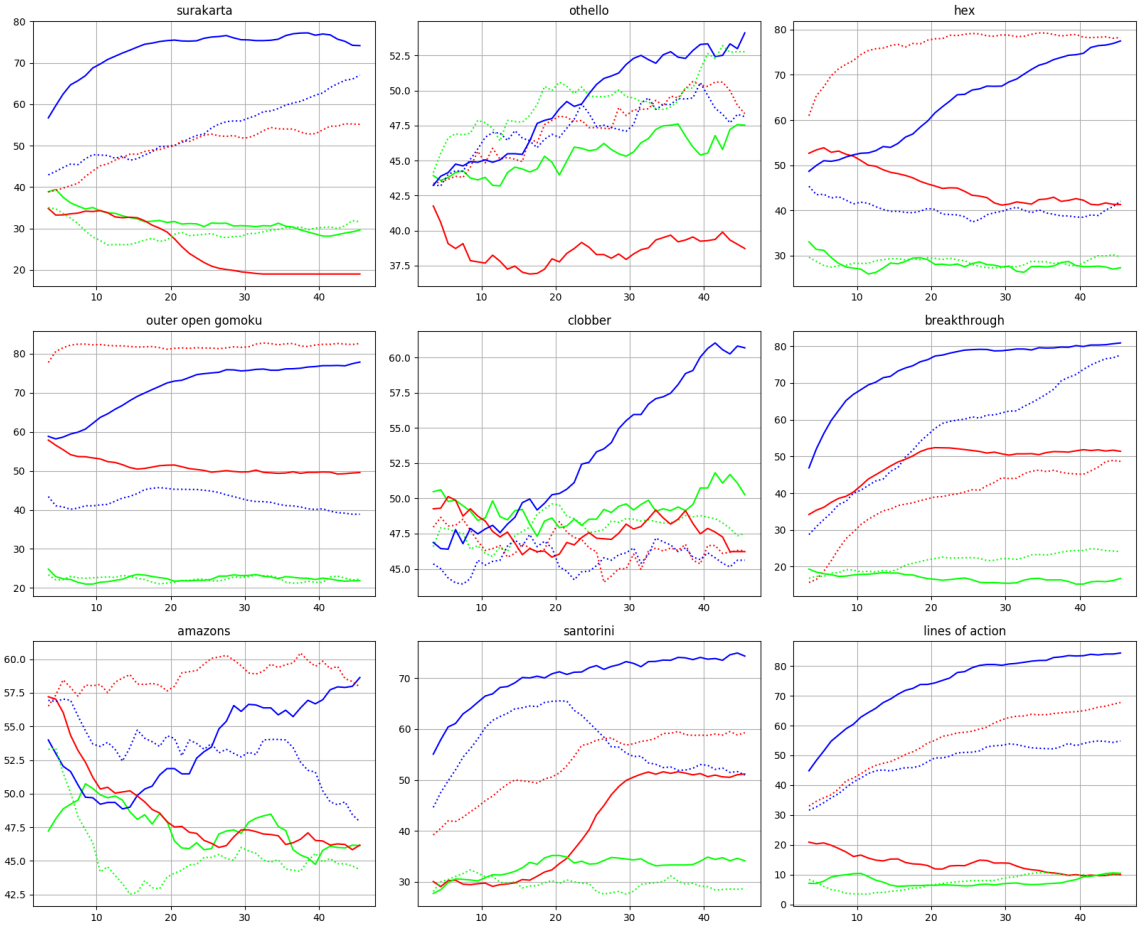

Figure 1: Evolutions of the winning percentages of the combinations of the experiment of Section 3.4, i.e. MCTS (dotted line) or iterative deepening alpha-beta (continuous line) with tree learning (blue line), root learning (red line), or terminal learning (green line). The display uses a simple moving average of 6 data.

purpose is to generate states for learning. With root or terminal learning, these states are not learned and therefore generated for nothing.

The idea of Descent is to combine UBFM with deterministic end-game simulations that provide informative values for learning.

The *Descent* algorithm recursively selects the best child of the current node, which becomes the new current node. More formally, in the current state s_k, the action a_k defined by the following formula is played:

$$a_k \leftarrow \begin{cases} \underset{a \in \texttt{actions}(s)}{\arg\max} \; v\left(s_k, a\right) & \text{if first\_player}\left(s_k\right) \\ \underset{a \in \texttt{actions}(s)}{\arg\min} \; v\left(s_k, a\right) & \text{otherwise} \end{cases}$$

The Descent algorithm performs this recursion from the root (the current state of the game), denoted s_0, until reaching a terminal node (an end game), denoted s_L. It thus constructs a sequence of states of length $L+1$ denoted by $\{s_k\}_{k=0}^{L}$. It then updates the minimax value of each selected node s_k in

Algorithm 9 Descent minimax algorithm (see Table 1 for the definitions of symbols ; note: S is the set of states which are non-leaves or terminal and $T = (v)$). .

Function `descent_iteration` (s, S, T, f_θ, f_t)
  if terminal (s) **then**
    $S \leftarrow S \cup \{s\}$
    $v(s) \leftarrow f_\mathrm{t}(s)$
  else
    if $s \notin S$ **then**
      $S \leftarrow S \cup \{s\}$
      foreach $a \in$ `actions` (s) **do**
        if terminal $(a\,(s))$ **then**
          $v(s,a) \leftarrow$ `descent_iteration` $(a(s), S, T, f_\theta, f_\mathrm{t})$
        else
          $v(s,a) \leftarrow f_\theta\,(a(s))$
    $a \leftarrow$ `best_action` (s)
    $v(s,a) \leftarrow$ `descent_iteration` $(a(s), S, T, f_\theta, f_\mathrm{t})$
    $a \leftarrow$ `best_action` (s)
    $v(s) \leftarrow v(s,a)$
  return $v(s)$

Function `best_action` (s)
  if `first_player` (s) **then**
    return $\underset{a \in \texttt{actions}(s)}{\arg\max} \; v\,(s,a)$
  else
    return $\underset{a \in \texttt{actions}(s)}{\arg\min} \; v\,(s,a)$

Function `descent` (s, S, T, f_θ, f_t, τ)
  $t =$ `time` $()$
  while `time` $() - t < \tau$ **do** `descent_iteration` (s, S, T, f_θ, f_t)
  return S, T

reverse order starting from s_{T-1} by the following formulas:

$$v(s_k, a_k) \leftarrow v\left(s_{k+1}\right)$$

$$v\left(s_k\right) \leftarrow \begin{cases} \max\limits_{a \in \texttt{actions}(s)} v\left(s_k, a\right) & \text{if first\_player}\left(s_k\right) \\ \min\limits_{a \in \texttt{actions}(s)} v\left(s_k, a\right) & \text{otherwise} \end{cases}$$

The Descent algorithm repeats this recursive operation starting from the root as long as there is some search time left. *Descent* is almost identical to UBFM. The only difference is that Descent performs an iteration until reaching a terminal state while UBFM performs this iteration until reaching a leaf of the tree (UBFM stops the iteration much earlier). In other words, during an iteration, UBFM just extends one of the leaves of the game tree while Descent recursively extends the best child from this leaf until reaching the end of the game.

Descent is formally described in Algorithm 9. To use Descent with tree learning, the method $\texttt{descent}(s, S, T, f_\theta, f_{\mathrm{t}}, \tau)$ must remplace $\texttt{search}(s, S, T, f_\theta, f_{\mathrm{t}}, \tau)$ in Algorithm 3.

The Descent algorithm shares the advantage of UBFM, namely performing deeper searches to identify better actions. By learning the values of the game tree (by using for example tree learning), it also has the advantage of a minimax search at depth 1, i.e. to raise the values of the terminal nodes to the other nodes more quickly. In addition, the states thus generated are closer to the terminal states. Their values therefore constitute more accurate approximations of the true minimax values.

Remark 11. In the experiments of this article, when there is a value tie in best_action(s), the tie is broken at random.

4.2 Comparison of Search Algorithms for Game Learning

We compare Descent with tree learning to MCTS (with root and tree learning), iterative deepening alpha-beta (with root and tree learning), and UBFM with tree learning, using the protocol of Section 3.3. There are a total of 6 combinations. The experiment was repeated 32 times. The winning percentage of a combination for each game and for each evaluation step (i.e. each hour) is therefore calculated from 384 matches (recall that there is a first player match and a second player match per pair of combinations).

The winning percentage curves are shown in Figure 2. The final winning percentages are shown in Table 3. Each value in the table is computed from $12,288$ matches. Descent attains the highest average performance across all games. For two games (Surakarta and Outer Open Gomoku), the difference with UBFM is small, but Descent still outperforms the traditional algorithms (MCTS and alpha-beta). For each game, Descent achieves the highest final winning percentage, except in Santorini, where it is 2% lower than UBFM, the best-performing algorithm for that game. Averaged across all games, Descent achieves an 82% win rate, exceeding UBFM, the second-best combination, by 18%. It also surpasses ID with tree learning, the third-best combination, by 34%.

In conclusion, Descent is the most effective search algorithm for learning state evaluation functions. UBFM (with tree learning) ranks second, at times approaching Descent's performance and at times more distant, but consistently outperforming the remaining algorithms.

5. Completion

In this section, we propose a complementary technique, called *completion*, which adjusts state evaluation functions by incorporating state resolution.

Remark 12. We do not evaluate this technique here in order to avoid unnecessary complexity and to focus on the learning aspects that constitute the main contributions of this paper. However, the evaluation of this technique's impact on performance is available in (Cohen-Solal, 2026a). In this study, the effectiveness of the completion mechanism has been extensively validated. In an evaluation encompassing 22 different games from the Computer Olympiad, the integration of completion into Unbounded Minimax search was shown to improve playing strength significantly. Across approximately 59,400 matches (with a search budget of 10 seconds per move), the mechanism yielded a mean score increase of 6.34%, with a 95% global confidence interval of [6.03%, 6.66%] obtained

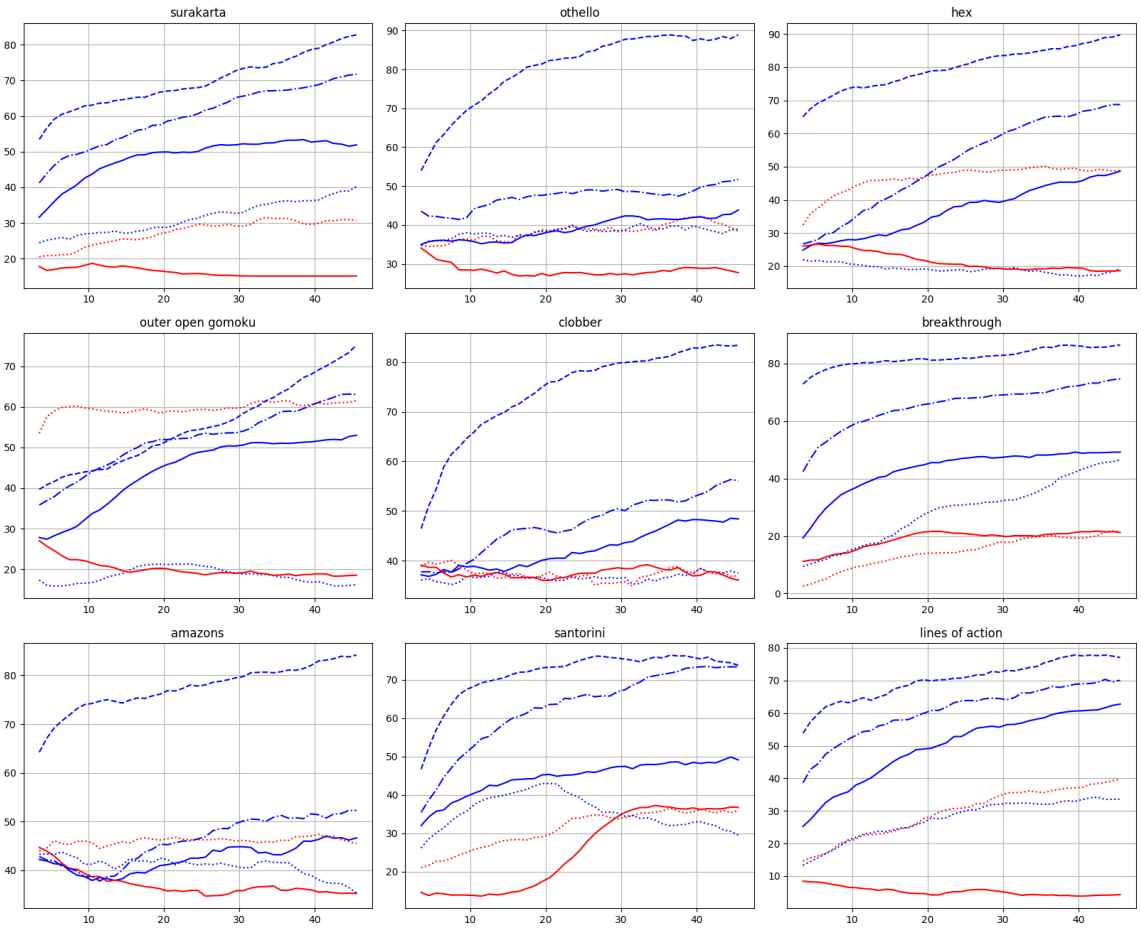

Figure 2: Evolutions of the winning percentages of the combinations of the experiment of Section 4.2, i.e. of Descent (dashed line), UBFM (dotted dashed line), MCTS (dotted line), and iterative deepening alpha-beta (continuous line) with tree learning (blue line) or root learning (red line). The display uses a simple moving average of 6 data.

via stratified bootstrapping. These results indicate that the benefits of state resolution propagation are robust across a wide variety of game structures.

5.1 Completion Concept

Relying solely on state values computed from the terminal *and* adaptive evaluation functions can lead to undesirable behaviors. More precisely, if we only seek to maximize the value of states, we will then choose to play a state s rather than another state s' when s is of greater value than s' even if s' is a winning resolved state (a state is *resolved* if we know the result of the match starting from that state in which the two players play optimally). A search algorithm can resolve a state. This happens when all the leaves of the subtree starting from that state are terminal. Choosing s over s', where s' is a winning resolved state, is suboptimal when s is either unresolved or not winning[6]. By choosing s, guarantee of winning is lost. The left graph of Figure 3 illustrates such a scenario.

It is therefore necessary to take into account both the value of states and the resolution of states.

5.2 Completed Algorithms

The completion technique proposed in this section provides one approach to addressing this issue. It consists, on the one hand, in associating with each state s a completion value $c(s)$ and a resolution value $r(s)$. A state is then resolved when its resolution value $r(s) = 1$, in which case its game-theoretical value is indeed proven and given by its completion value $c(s)$. The completion value $c(s)$ of a leaf state s is 0 if the state s is not terminal or if s is a draw, 1 if s is a winning terminal state, and -1 if s is a losing terminal state. The value $c(s)$ of a non-leaf state s is computed as the minimax value of the subtree of the partial game tree starting from s where the leaves are evaluated by their completion value. The resolution value $r(s)$ of a leaf state s is 0 if the state s is not terminal and 1 if it is terminal. The resolution value $r(s)$ of a non-leaf state s is 1 if $|c(s)| = 1$. Otherwise, $r(s)$ is the minimum of the resolution values of the childen of s.

6. There is perhaps, in certain circumstances, an interest in making this error from the point of view of learning.

	tree learning				root learning	
	Descent	UBFM	MCTS	ID	MCTS	ID
Othello	89.4%	47.2%	37.7%	42.7%	44.9%	22.1%
Hex	94.9%	71.7%	20.5%	50.7%	50.2%	20.5%
Clobber	83.0%	56.7%	32.9%	48.9%	42.0%	35.5%
Outer Open Gomoku	77.6%	63.9%	18.0%	51.6%	64.7%	18.2%
Amazons	84.3%	55.3%	32.5%	46.3%	43.7%	31.7%
Breakthrough	86.5%	72.5%	45.5%	47.6%	19.2%	21.0%
Santorini	69.9%	71.8%	31.9%	47.6%	37.7%	40.2%
Surakarta	82.8%	69.7%	41.2%	42.1%	29.4%	14.5%
Lines of Action	73.4%	66.9%	36.1%	57.4%	39.4%	3.7%
mean	82.4%	64.0%	32.9%	48.3%	41.2%	23.1%

Table 3: Final winning percentages of the combinations of the experiment of Section 4.2 (ID: iterative deepening alpha-beta ; see Section 3.3).

To calculate these values in practice, particularly when using transposition tables, we employ the following variables: $c(s,a)$ stores the completion value of the state reached by taking action a in state s, and is updated each time a is taken in s; it is not affected by subsequent changes to that state's value. In addition, $r(s,a)$ stores the resolution value of the state reached by taking action a in state s, and is updated each time a is taken in s; it is not affected by subsequent changes to that state's value. These are local, not global, state value estimates.

Remark 13. Thus, we do not necessarily have $c(s,a) = c\left(a\left(s\right)\right)$. This equality may be violated if the state $a(s)$ is later revisited without passing through s, even when using transpositions.

With completion, states are compared using pairs $\left(c\left(\cdot\right), v\left(\cdot\right)\right)$ under lexicographic ordering, rather than using $v\left(\cdot\right)$ alone.

Completion consists of three key points:

1. During the search: c, v, and r are propagated. The values of a state s is computed in calculating $(c(s), v(s))$ as the minimax value of the subtree of the partial game tree starting from s where the leaves l are evaluated by $(c(l), v(l))$. Thus, the value $v\left(s\right)$ of a winning state s is always the value of the corresponding winning terminal leaf. More formally, completion propagation consists in applying the following update rules once a state s has been analyzed, i.e., when one of the actions of s, denoted a^*, is played from s during the search:

$$v\left(s, a^*\right) \leftarrow v\left(a^*\left(s\right)\right)$$

$$c\left(s, a^*\right) \leftarrow c\left(a^*\left(s\right)\right)$$

$$r\left(s, a^*\right) \leftarrow r\left(a^*\left(s\right)\right)$$

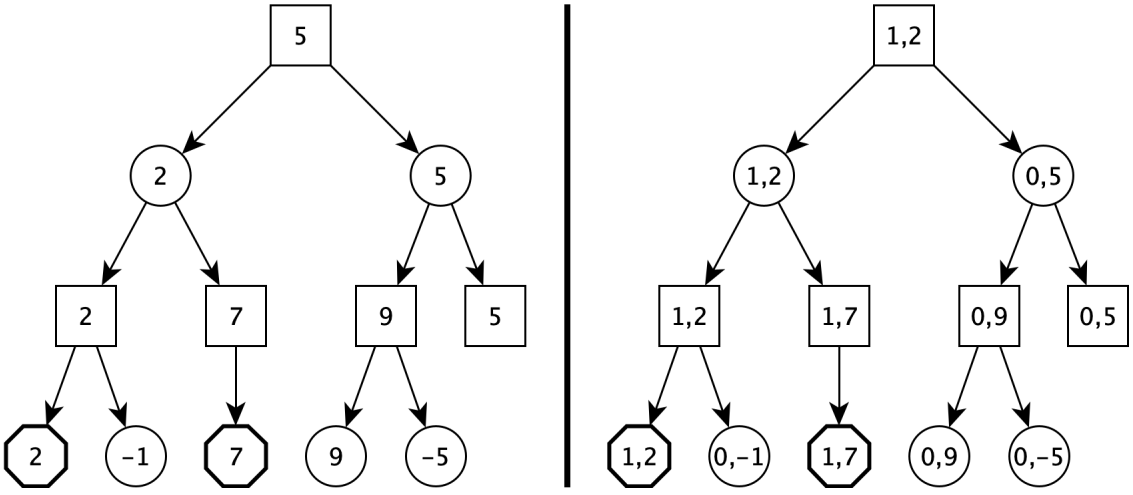

Figure 3: The left graph shows a game tree in which maximizing does not yield the optimal decision. The right graph represents the same tree after completion, with nodes labeled by pairs of values; here, maximizing results in the best decision. Node shapes indicate type: square for first-player (max) nodes, circle for second-player (min) nodes, and octagon for terminal nodes.

$$(c(s), v(s)) \leftarrow \begin{cases} \max\limits_{a \in \texttt{actions}(s)} (c(s,a), v(s,a)) & \text{if first\_player}(s) \\ \min\limits_{a \in \texttt{actions}(s)} (c(s,a), v(s,a)) & \text{otherwise} \end{cases}$$

$$r(s) \leftarrow \begin{cases} 1 & \text{if } |c(s)| = 1 \\ \min_{a \in \texttt{actions}(s)} r(s,a) & \text{otherwise} \end{cases}$$

During the initial analysis of s, we have for all $a \in \texttt{actions}(s)$:

$$v(s,a) \leftarrow \begin{cases} f_{\mathrm{t}}(a(s)) & \text{if terminal}(a(s)) \\ f_{\theta}(a(s)) & \text{otherwise} \end{cases}$$

$$c(s,a) \leftarrow \begin{cases} \mathrm{b}_{\mathrm{t}}(a(s)) & \text{if terminal}(a(s)) \\ 0 & \text{otherwise} \end{cases}$$

$$r(s,a) \leftarrow \text{terminal}(a(s))$$

2. During the search: resolved states are never played. To decide which action to play during the search, we choose the best unresolved action if it exists and otherwise the best resolved action (i.e. we choose the action a which maximize $(-r(s,a), c(s,a), v(s,a), -n(s,a))$ in a max state and which minimize $(r(s,a), c(s,a), v(s,a), n(s,a))$ in a min state).

3. Post-search: c and r are used to always play an action leading to a winning resolved state and never play an action leading to a losing resolved state if possible; otherwise, the standard action selection method is used. Thus, if among the available actions we know that one of the actions is winning, we play it. If there is none, we play according to the chosen action selection method among the actions not leading to a losing resolved state (if possible). We call this subtechnique *completed action selection*. More formally, the first player plays the following action a^*:

$$a^* \leftarrow \begin{cases} \arg\max\limits_{a \in A_+} v(s,a) & \text{if } |A_+| > 0 \\ \text{action\_selection}(s, A_0) & \text{else if } |A_0| > 0 \\ \arg\max\limits_{a \in \texttt{actions}(s)} v(s,a) & \text{else} \end{cases}$$

where the method $\text{action\_selection}(s, A)$ returns an action from A to play in s, $A_+ = \{a \in \texttt{actions}(s) \,|\, c(s,a) > 0\}$, and $A_0 = \{a \in \texttt{actions}(s) \,|\, c(s,a) = 0\}$. And, the second player plays the following action a^*:

$$a^* \leftarrow \begin{cases} \arg\min\limits_{a \in A_-} v(s,a) & \text{if } |A_-| > 0 \\ \text{action\_selection}(s, A_0) & \text{else if } |A_0| > 0 \\ \arg\min\limits_{a \in \texttt{actions}(s)} v(s,a) & \text{else} \end{cases}$$

with $A_- = \{a \in \texttt{actions}(s) \,|\, c(s,a) < 0\}$.

The right graph of Figure 3 illustrates state labeling performed by completion.

Without completion, algorithms such as Descent or Unbounded Minimax can get stuck in a non-optimal fixed point. The completion technique ensures that with sufficient thinking time, the optimal strategy is found (Cohen-Solal, 2026a, 2021). More precisely, completion is sound for finite perfect information games that are correctly defined, i.e. whose complete game tree is a finite lower semilattice.

Moreover, exploiting state resolution enables *resolution stop*, i.e. early termination of search in resolved subtrees, thereby reducing computational cost. Descent algorithm modified to use the completion and the resolution stop is described in Algorithm 10. Unbounded Minimax algorithm modified to use the completion and the resolution stop is described in Algorithm 13.

Algorithm 10 *Descent* tree search algorithm with completion and resolution stop (see Table 1 for the definitions of symbols and Algorithm 11 for the definitions of completed_best_action(s), completed_best_action_dual(s) and backup_resolution(s)). Note: $T = (v, c, r, n)$ and S is the set of states of the partial game tree which are non-leaves or terminal.

Function `descent_iteration` (s, S, T, f_θ, f_t)
- **if** $\mathrm{terminal}(s)$ **then**
  - $S \leftarrow S \cup \{s\}$
  - $r(s), c(s), v(s) \leftarrow 1, \mathrm{b_t}(s), f_\mathrm{t}(s)$
- **else**
  - **if** $s \notin S$ **then**
    - $S \leftarrow S \cup \{s\}$
    - **foreach** $a \in$ `actions` (s) **do**
      - **if** $\mathrm{terminal}(a(s))$ **then**
        - $r(s,a), c(s,a), v(s,a) \leftarrow$ `descent_iteration` $(a(s), S, T, f_\theta, f_\mathrm{t})$
      - **else**
        - $r(s,a), c(s,a), v(s,a) \leftarrow 0, 0, f_\theta(a(s))$,
    - $a_b \leftarrow$ `completed_best_action` (s, `actions`(s))
    - $c(s), v(s) \leftarrow c(s,a_b), v(s,a_b)$
    - $r(s) \leftarrow$ `backup_resolution` (s)
  - **if** $r(s) = 0$ **then**
    - $A \leftarrow \{a \in$ `actions`$(s) \mid r(s,a) = 0\}$
    - $a \leftarrow$ `completed_best_action_dual` (s, A)
    - $n(s,a) \leftarrow n(s,a) + 1$
    - $r(s,a), c(s,a), v(s,a) \leftarrow$ `descent_iteration` $(a(s), S, T, f_\theta, f_\mathrm{t})$
    - $a \leftarrow$ `completed_best_action` (s, `actions`(s))
    - $c(s), v(s) \leftarrow c(s,a), v(s,a)$
    - $r(s) \leftarrow$ `backup_resolution` (s)
- return $r(s), c(s), v(s)$

Function `completed_descent` (s, S, T, f_θ, f_t, τ)
- $t =$ `time()`
- **while** `time()` $- t < \tau \wedge r(s) = 0$ **do** `descent_iteration` (s, S, T, f_θ, f_t)
- return S, T

Remark 14. Tie breaking is decided randomly.

Remark 15. It is not clear, however, that *completed action selection* improves, in practice, performance as it prunes a portion of the game tree whose values could be useful for learning (though it would be surprising if that were not the case).

Remark 16. In some application cases, we will prefer to ensure the draw rather than trying to win. Algorithm 13 must then be adapted to decide the action to play. If the first player prefers to guarantee draws, it must instead maximize:

$$(c(s,a), r(s,a), n(s,a), v(s,a)).$$

If the second player prefers to guarantee draws, it must instead minimize:

$$(c(s,a), -r(s,a), -n(s,a), v(s,a)).$$

Remark 17. For a game where there is no draw, the computation of the resolution value is not necessary (all necessary information is in the completion value).

Remark 18. For games without draw, the completion of Unbounded Minimax can be simply implemented by giving a value of $+\infty$ to winning terminal states and a value of $-\infty$ to losing terminal states (that is to say, it is sufficient to replace the terminal function by this infinite terminal function). However, this may lead to aberrant but harmless behavior, such as playing unnecessarily long winning strategies and this does not generalize to stochastic games.

Moreover, in the additional special case of adaptive evaluation functions with values in the open interval $(-1, 1)$, it is not sufficient that the terminal evaluation functions are in $\{-1, 1\}$. Indeed, in

Algorithm 11 Definition of the algorithms completed_best_action(s, A), which computes the *a priori* best action by using completion, and backup_resolution(s), which updates the resolution of s from its child states.

Function `completed_best_action(`s, A`)`
- **if** `first_player(`s`)` **then**
  - return $\arg\max_{a \in A} \left(c(s,a), v(s,a), n\left(s,s'\right)\right)$
- **else**
  - return $\arg\min_{a \in A} \left(c(s,a), v(s,a), -n\left(s,s'\right)\right)$

Function `completed_best_action_dual(`s, A`)`
- **if** `first_player(`s`)` **then**
  - return $\arg\max_{a \in A} \left(c(s,a), v(s,a), -n\left(s,s'\right)\right)$
- **else**
  - return $\arg\min_{a \in A} \left(c(s,a), v(s,a), n\left(s,s'\right)\right)$

Function `backup_resolution(`s`)`
- **if** $|c(s)| = 1$ **then**
  - return 1
- **else**
  - return $\min_{a \in \texttt{actions}(s)} r(s,a)$

practice, with rounding errors, the adaptive evaluation function being real value will have values in $[-1, 1]$.

Remark 19. There is some similarity between Completion and MCTS solver (Winands, Björnsson, and Saito, 2008) in that both propagate terminal information during backups to establish the game-theoretical value of states; however, their objectives and underlying mechanisms differ substantially.

First, completion is able to prove draws, whereas MCTS solver does not provide such guarantees. Second, MCTS solver may select a proven losing move after the search when unresolved alternatives remain. In addition, incorporating resolved states into MCTS introduces a bias in the search; to mitigate this, MCTS solver relies on a visit-count threshold to decide when proven values should influence the policy. In contrast, completion does not rely on such heuristics: it integrates solved information in a principled way without introducing this type of bias, and does not require any threshold mechanism.

Algorithm 12 $\mathrm{UBFM_s}$ tree search algorithm with completion and resolution stop (see Table 1 for the definitions of symbols and Algorithm 11 for the definitions of completed_best_action(s) and backup_resolution(s)). Note: $T = (v, c, r, n)$. Note: Adding terminal states in S is only useful during training with tree learning, so it should not be done during confrontations (i.e. evaluation matches).

Function `ubfms_iteration` (s, S, T, f_θ, f_t)
- **if** terminal (s) **then**
  - $S \leftarrow S \cup \{s\}$
  - $r(s), c(s), v(s) \leftarrow 1, \mathrm{b_t}(s), f_\mathrm{t}(s)$
- **else**
  - **if** $r(s) = 0$ **then**
    - **if** $s \notin S$ **then**
      - $S \leftarrow S \cup \{s\}$
      - **foreach** $a \in$ `actions(`s`)` **do**
        - **if** terminal $(a(s))$ **then**
          - $r(s,a), c(s,a), v(s,a) \leftarrow$ `ubfms_iteration` $(a(s), S, T, f_\theta, f_\mathrm{t})$
        - **else**
          - $r(s,a), c(s,a), v(s,a) \leftarrow 0, 0, f_\theta(a(s))$
    - **else**
      - $A \leftarrow \{a \in \mathtt{actions}(s) \mid r(s,a) = 0\}$
      - $a \leftarrow$ `completed_best_action_dual` (s, A)
      - $n(s,a) \leftarrow n(s,a) + 1$
      - $r(s,a), c(s,a), v(s,a) \leftarrow$ `ubfms_iteration` $(a(s), S, T, f_\theta, f_\mathrm{t})$
    - $a \leftarrow$ `completed_best_action` (s, `actions`(s))
    - $c(s), v(s) \leftarrow c(s,a), v(s,a)$
    - $r(s) \leftarrow$ `backup_resolution` (s)
- return $r(s), c(s), v(s)$

Function `ubfms_tree_search` (s, S, T, f_θ, f_t, τ)
- $t =$ `time()`
- **while** `time()` $- t < \tau \wedge r(s) = 0$ **do** `ubfms_iteration` (s, S, T, f_θ, f_t)
- return S, T

Remark 20. An experimental study on the individual and combined impacts of various enhancements to Unbounded Minimax, including completion and transposition tables, is available here (Cohen-Solal and Cazenave, 2025b).

6. Reinforcement Heuristic to Improve Learning Performance

In this section, we propose the technique of *reinforcement heuristic*, which consists of replacing the classical terminal evaluation function, the game gain – that we denote by $\mathrm{b_t}$, which returns 1 if the first player wins, -1 if the second player wins, and 0 in case of a draw (Young et al., 2016; Silver et al., 2017b; Gao et al., 2018) – by another heuristic to evaluate terminal states during the learning process. Using reinforcement heuristics alters the evaluation of non-terminal states through minimax propagation, leading to different partial game trees and learning trajectories, which may affect performance. We first define the notion of a reinforcement heuristic and then introduce several such heuristics. Finally, we compare the reinforcement heuristics that we propose to the classical terminal evaluation function.

6.1 Reinforcement Heuristic Definition

A reinforcement heuristic is a terminal evaluation function that is more expressive than the classical terminal function, i.e. the game gain.

Definition 21. A reinforcement heuristic h_r is a function that preserves the order of the game gain function $\mathrm{b_t}$: for any two terminal states of the game s, s', $\mathrm{b_t}(s) < \mathrm{b_t}(s')$ implies $h_r(s) < h_r(s')$.

6.2 Some Reinforcement Heuristics

In the following subsections, we propose different reinforcement heuristics.

6.2.1 SCORING

Some games have a natural reinforcement heuristic: the game score. For example, in the case of the game Othello (and in the case of the game Surakarta), the game score is the number of a player's pieces minus the number of the opponent's pieces (the goal of the game is to have more pieces than

Algorithm 13 $\mathrm{UBFM_s}$ action decision algorithm with completion (see Algorithm 12 for the definition of the method ubfms_tree_search() ; see Table 1 for the definitions of symbols). Note: $T = (v, c, r, n)$ and S is the set of states of the game tree which are non-leaves or terminal.

Function `safest_action(`*s, T*`)`
  if `first_player(`*s*`)` **then**
    return $\underset{a \in \texttt{actions}(s)}{\arg\max}\ (c(s,a), n(s,a), v(s,a))$
  else
    return $\underset{a \in \texttt{actions}(s)}{\arg\min}\ (c(s,a), -n(s,a), v(s,a))$

Function `ubfms(`*s, S, T,* f_θ, f_{t}, τ`)`
  $S, T \leftarrow$`ubfms_tree_search(`*s, S, T,* f_θ, f_{t},τ`)`
  return `safest_action(`*s, T*`)`

its opponent at the end of the game). The scoring function used as a reinforcement heuristic consists of evaluating the terminal states by the final score of the game. With that reinforcement heuristic, the adaptive evaluation function seeks to learn the score of states. In the context of an algorithm based on minimax, the score of a non-terminal state is the minimax value of the subtree starting from that state whose terminal leaves are evaluated by their scores. After training, the adaptive evaluation function then contains more information than just an approximation of the result of the game, it contains an approximation of the score of the game. If the game score is *intuitive*, the scoring heuristic should improve learning performances.

Remark 22. In the context of the game of the Amazons, the score is the size of the territory of the winning player, i.e. the squares which can be reached by a piece of the winning player. It is approximately the number of empty squares.

6.2.2 ADDITIVE AND MULTIPLICATIVE DEPTH HEURISTICS

Now we offer the following reinforcement heuristic: the *depth heuristic*. It assigns higher values to winning states that are closer to the start of the game than to those occurring later. Reinforcement learning with the depth heuristic is learning the duration of matches in addition to their results. This learned information is then used to try to win as quickly as possible and try to lose as late as possible. If learning durations can be done, this should reduce the length of games and therefore speed up learning (by increasing the number of matches played in the same time but also by reducing the propagation time of end-of-game information to the start-of-game states). Seeking to lose for as long as possible helps avoid missing out on potential long-term wins (even if it makes games last longer).

We propose two realizations of the depth heuristic: the *additive depth heuristic*, that we denote by $\mathrm{p_t}$, and the *multiplicative depth heuristic*, that we denote by $\mathrm{p_t}'$. The evaluation function $\mathrm{p_t}$ returns the value l if the first player wins, the value $-l$ if the second player wins, and 0 in case of a draw, with $l = P - p + 1$ where P is the maximum number of playable actions in a match and p is the number of actions played since the beginning of the current match. For the game of Hex, l is the number of empty cells on the board plus 1. For the games where P is very large or difficult to compute, we can instead use $l = \max\left(1, \tilde{P} - p\right)$ with $\tilde{P}$ a constant approximating P (close to the empirical maximum length of matches). The evaluation function $\mathrm{p_t}'$ is identical except that l satisfies $l = \frac{P'}{p}$, and P' can be the exact or empirical average length of matches.

Remark 23. Note that the idea of fast victory and slow defeat has already been proposed but not used in a learning process (Cazenave, Saffidine, Schofield, and Thielscher, 2016).

6.2.3 CUMULATIVE MOBILITY

The next reinforcement heuristic that we propose is *cumulative mobility*. It favors matches in which each player tries to maximize their number of available actions while minimizing those of their opponent. The implementation used in this article is as follows. The value of a terminal state is $\frac{M_1}{M_2}$ if the first player wins, $-\frac{M_2}{M_1}$ if the second player wins, and 0 in case of a draw, where M_1 is the mean of the number of available actions in each turn of the first player since the start of the game and M_2 is the mean of the number of available actions in each turn of the second player since the start of the game.

6.2.4 PIECE COUNTING: PRESENCE

Finally, we propose as reinforcement heuristic: the *presence* heuristic. It consists in taking into account the number of pieces of each player. This heuristic is based on the principle that the more a player has pieces the more this one has an advantage. There are several implementations for the presence heuristic. We use in this article the following implementation: the heuristic value is $\max(n_1 - n_2, 1)$ if the first player wins, $\min(n_1 - n_2, -1)$ if the second player wins, and 0 in case of a draw, where n_1 is the number of pieces of the first player and n_2 is the number of pieces of the second player. Note that in the games Surakarta and Othello, the score corresponds to a presence heuristic.

6.3 Comparison of Reinforcement Heuristics

We now compare the different heuristics that we have proposed to the classical terminal evaluation function $\mathrm{b_t}$ on different games, using the protocol of Section 3.3. Each combination uses Descent with completion (Algorithm 10), completed ϵ-greedy (see Algorithm 2 and Section 5). Each combination uses a different terminal evaluation function. These terminal evaluations are the classical game gain evaluation function $\mathrm{b_t}$, the additive depth heuristic, the multiplicative depth heuristic, the scoring heuristic, the cummulative mobility, and the presence heuristic. Other parameters are the same as Section 3.4. There are, at most, a total of 6 combinations per game (on some games, some heuristics are not evaluated because they are trivially of no interest or equivalent to another heuristic). The experiment was repeated 48 times. The winning percentage of a combination for each game and for each evaluation step (i.e. each hour) is therefore calculated from 288 to 576 matches. The winning percentage curves are shown in Figure 4.

The final winning percentages are shown in Table 4. Each percentage of the table has required between $13,824$ and $27,648$ matches. On average and for 7 of the 9 games, the classic terminal heuristic has the worst percentage (exceptions are Othello and Lines of Action). In scoring games, scoring is the best heuristic, as we might expect. Leaving aside the score heuristic, with the exception of Surakarta, Othello and Clobber, it is one of the two depth heuristics that has the best winning percentage. In Surakarta and Clobber, mobility is just ahead of the depth heuristics. On average, using the additive depth heuristic instead of using the classic evaluation increases the winning percentage by 15%, and using the best depth heuristic increases the winning percentage by 19%. The final percentages summarize the curves quite well. Note that the positive impact of the additive depth heuristic compared to the other heuristics (except score) is particularly clear at Breakthrough, Amazons, Hex, and Santorini. In a similar manner, the positive impact of the multiplicative depth heuristic is particularly clear at Clobber, Hex, and Outer Open Gomoku.

In conclusion, the use of generic reinforcement heuristics has significantly improved performances and the depth heuristics are prime candidates as a powerful generic reinforcement heuristic.

Remark 24. Reinforcement heuristics allow injecting knowledge at different levels (general, semi-general, or domain-specific). In our view, scoring—when explicitly defined by the game rules—constitutes general information rather than domain-specific knowledge.

Empirically, the best results are obtained using scoring when available, and the depth heuristic otherwise. This keeps the approach broadly general in practice. Moreover, when restricting to purely generic reinforcement heuristic (with the depth heuristics), performance remains strong across all tested games, with the notable exception of Othello.

While incorporating features such as mobility can further improve performance, these fall into semi-general or domain-specific enhancements and are not required for the baseline results. Thus, the "without knowledge" claim should be understood as "without domain-specific knowledge", as the core method relies only on general or trivially derived rule-based information.

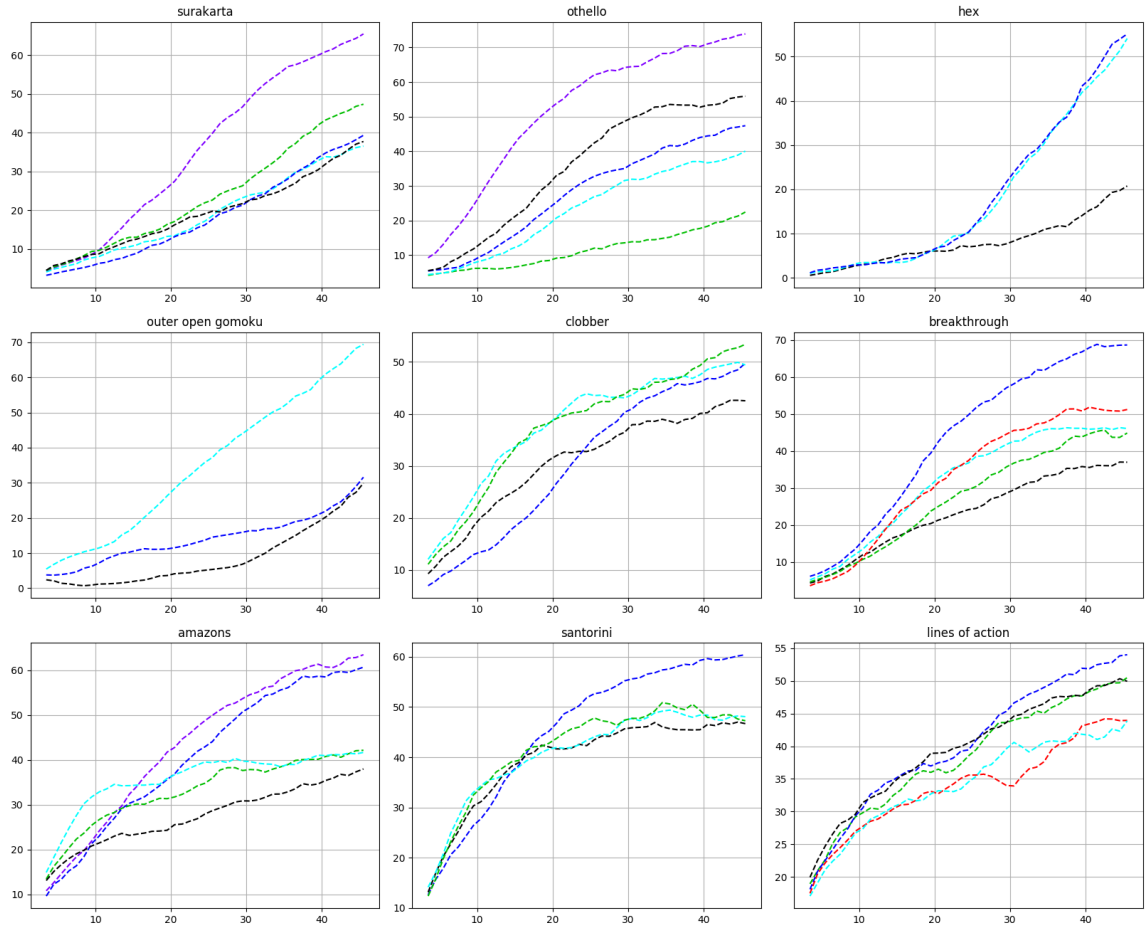

Figure 4: Evolutions of the winning percentages of the combinations of the experiment of Section 6.3, i.e. of the use of the following heuristics: classic (black line), score (purple line), additive depth (blue line), multiplicative depth (turquoise line), cumulative mobility (green line), and presence (red line). The display uses a simple moving average of 6 data.

7. Ordinal Distribution

In this section, we propose another technique, a new action selection distribution. It is based on a new probability distribution, which we call the *ordinal distribution*.

The ordinal distribution does not depend on the value of states. However, it depends on the order of their values. On the one hand, this distribution has the advantage of providing different

probability distributions for actions that are not the best, unlike ϵ-greedy. On the other hand, it has a more intuitive interpretation than the softmax function. Under this distribution, the best action is selected with probability p. If we do not choose it, we choose the second best action with probability approximately p. Otherwise, we choose the third action with probability approximately p, etc. More precisely, with this distribution, the conditional probability of playing an action knowing that we do not play a better action is a linear interpolation between the uniform distribution and the max distribution (with respect to the exploitation parameter ϵ'). The ordinal distribution formula is:

$$P(c_i) = \left(\epsilon' + \frac{1-\epsilon'}{n-i}\right) \cdot \left(1 - \sum_{j=0}^{j<i} P(c_j)\right)$$

with n the number of children of the root, $i \in \{0, \ldots, n-1\}$, c_i the i-th best child of the root, $P(c_i)$ the probability of playing the action leading to the child c_i, and $\epsilon' \in [0, 1]$ the exploitation parameter ($\epsilon' = 1 - \epsilon$). Algorithm 14 describes the action selection method resulting from the use of the ordinal distribution with an optimized calculation.

Remark 25. We experimentally compared the ordinal distribution to theϵ-greedy distribution and to the softmax distribution on many games. The ordinal distribution performs better on average and on the majority of games than the other two distributions (the gain is however quite slight). However, we do not show these results in this article. These additional results will be presented in a separate future publication.

Remark 26. We also noticed that the ordinal distribution was easier to tune than the softmax distribution. In particular, the best parameters for the different games studied are close, unlike what happens for the softmax distribution.

			depth			
	classic	score	additive	multiplicative	mobility	presence
Othello	49.8%	70.6%	50.1%	48.9%	18.5%	score
Hex	33.3%	X	66.1%	60.4%	X	X
Clobber	43.7%	X	47.0%	49.8%	53.5%	X
Outer Open Gomoku	33.0%	X	41.4%	74.4%	X	X
Amazons	36.8%	67.9%	60.0%	50.7%	49.0%	X
Breakthrough	39.0%	X	69.5%	40.4%	43.9%	48.5%
Santorini	42.7%	X	59.7%	46.6%	43.3%	X
Surakarta	33.5%	68.9%	43.1%	35.3%	55.6%	score
Lines of Action	50.9%	X	57.1%	46.8%	53.7%	44.0%
mean	40.3%	69.1%	54.9%	50.4%	45.4%	46.3%

Table 4: Final winning percentages of the combinations of the experiment of Section 6.3 (X: heuristic without interest in this experiment or for the associated game ; presence coincides with score in Surakarta and Othello.

8. Comparison with ExIt

In this section, we compare ExIt and Athénan. We start by presenting the ExIt algorithm (Section 8.1). Thereafter, we compare the algorithmic differences between ExIt and Athénan (Section 8.2.) Next, we present the experiment carried out (Section 8.3) and its technical details (Section 8.4). Finally, we present the results of this experiment (Section 8.5).

8.1 The ExIt Algorithm

The ExIt (Expert Iteration) algorithm consists of using an expert algorithm to generate data and using an apprentice algorithm that learns that data by supervision to imitate the expert. In order for the process to improve, the expert must use the apprentice as the basis of its reasoning. The procedure can then be applied in a self-improving loop: the apprentice imitates the expert to improve, the expert uses the apprentice to improve, thereby providing better quality data, etc. In practice, the expert is a search algorithm and in particular a modified version of MCTS. The apprentice is a neural network used to evaluate states and provide a policy. The policy consists of probabilities of playing actions. It is used by the modified MCTS. The general algorithm of ExIt is given in Algorithm 15. To initialize the self-improvement procedure, the apprentice is trained on data generated using standard MCTS, i.e., without any prior knowledge.

The data generation strategy is as follows. Perform self-play matches to obtain the states to be evaluated by the expert serving as learning targets. In order not to have correlated data, only one game state per match is used. And so that it is not too costly, it is the apprentice which is used to perform the self-play matches. One of the states of each match will then be analyzed by the expert in order to label this data.

At the beginning of the ExIt process, only the policy is learned and used. The policy loss, called tree-policy target, used for training the policy network is given by the following formula:

$$-\sum_{a} \frac{n_{s,a}}{\sum_{a'} n_{s,a'}} \log \pi\left(a|s\right)$$

where $n_{s,a}$ is the number of times the action a is selected during the search in the state s, and $\pi\left(a|s\right)$ is the probability of playing a in s according to the neural network. The modified UCT value of the

Algorithm 14 Ordinal action distribution algorithm with simulated annealing ($\epsilon' = \frac{t}{t_{\max}}$) used in the experiments of this article (see Table 1 for the definitions of symbols).

Function `ordinal(`s`,` v`)`
- **if** `first_player(`s`)` **then**
  - $A \leftarrow$ `actions(`s`)` sorted in descending order by $a \mapsto v(s,a)$
- **else**
  - $A \leftarrow$ `actions(`s`)` sorted in ascending order by $a \mapsto v(s,a)$
- $i \leftarrow 0$
- $n \leftarrow |A|$
- **for** $a \in A$ **do**
  - **if** $\text{random}\left(\right) \leq \left(\frac{t}{t_{\max}} \cdot (n-i-1)+1\right) / (n-i)$ **then**
    - return a
  - $i \leftarrow i+1$

MCTS part of ExIt is given by the following formula:

$$\mathrm{UCT}_{\mathrm{ExIt}}(s,a) = \mathrm{UCT}(s,a) + w_a \cdot \frac{\pi(a|s)}{n_{s,a}+1}$$

where $\mathrm{UCT}(s,a)$ is the classical UCT term of MCTS, which manages the exploration exploitation dilemma (Coulom, 2007), and w_a is a constant.

Later during the training, the value network is learned and used in addition to the policy network. The two losses are thus added for the rest of the training. The value loss, referred to as the KL loss, used to train the value network is defined as follows:

$$-z \cdot \log(v(s)) - (1-z) \cdot \log(1-v(s))$$

where z is the result (classic gain) of the match and $v(s)$ is the value of the state s according to the neural network The modified UCT value of the MCTS part of ExIt when the value network is used is given by the following formula:

$$\mathrm{UCT}_{\mathrm{ExIt}}(s,a) = \mathrm{UCT}(s,a) + w_a \cdot \frac{\pi(a|s)}{n_{s,a}+1} + w_v \cdot \hat{Q}(s,a)$$

where w_v is a constant, and $\hat{Q}(s,a)$ is the backed up average of the network value after playing a in the state s during the search.

Algorithm 15 Algorithm ExIt (online Expert Iteration).

$\hat{\pi}_0 \leftarrow \mathrm{initial\_policy}()$
$\pi_0^* \leftarrow \mathrm{build\_expert}(\hat{\pi}_0)$
foreach $i \in \{1, \ldots, \mathrm{max\_iterations}\}$ **do**
  $S_i \leftarrow \mathrm{sample\_self\_play}(\hat{\pi}_{i-1})$
  $D_i \leftarrow \left\{ \left(s, \mathrm{imitation\_learning\_target}\left(\pi_{i-1}^*(s)\right)\right) \mid s \in S_i \right\}$
  $\hat{\pi}_i \leftarrow \mathrm{train\_policy}\left(\bigcup_{j \leq i} D_j\right)$
  $\pi_i^* \leftarrow \mathrm{build\_expert}(\hat{\pi}_{\mathrm{i}})$

8.2 Algorithmic Comparison between ExIt, AlphaZero and Athénan

In this section, we compare the following algorithms with each other: Athénan, ExIt, and also AlphaZero. We perform a detailed comparison and then provide a summary.

8.2.1 Detailed Algorithmic Comparison

AlphaZero and Athénan share several commonalities that distinguish them from ExIt. We begin by detailing the characteristics shared by Athénan and AlphaZero that distinguish them from ExIt. On the one hand, unlike ExIt, the same neural network is kept from the start to the end of the learning process (it is not reset before each learning phase). Experience replay is used (although, at least in the context of Athénan, this is optional: it significantly improves learning performances but learning also works without it). ExIt uses the *early stopping technique* and, in its best version, relearns all the data generated from the beginning at each learning phase (performing the learning phases only with the last data gave inferior results during their experiments).

Furthermore, ExIt does a "warm-start": it performs a pre-training by learning the data of matches generated from base MCTS (base MCTS does not use any learned policy or value function: the leaf states are evaluated only by statistics of victory of random games). Then, ExIt only uses and learns the policy with its modified MCTS. Finally, it uses and learns the policy and the value of states with its modified MCTS. It learns in this way more quickly at the beginning of the training. Athénan performs a pre-training by learning the data of random terminal states. Similar to AlphaZero, Athénan uses mean square error for learning (not the KL loss).

On the other hand, unlike ExIt, with Athénan and AlphaZero, a search is performed in each state of the match to determine the move to play. Moreover, with AlphaZero and Athénan, the data of each state of the match is used during the learning phases. On the contrary, ExIt plays matches without search according to the policy of the neural network in order to perform more matches. In addition, ExIt performs a search for only one state of the match. It therefore uses only one data item per match for learning.

We now describe the features that differ between Athénan and ExIt and which are similar between ExIt and AlphaZero. During confrontations (evaluation matches), Athénan uses Unbounded Minimax with Safe Decision: $\mathrm{UBFM_s}$. ExIt and AlphaZero use MCTS. Unlike AlphaZero and ExIt, Athénan learns all data generated during search, i.e., the data from the partial game tree. In other words, for learning value of states, tree learning is used instead of terminal learning (thus much more than one piece of data is learned per search). Therefore, there is no need for large parallelization to generate a significant number of data for training (as done for ExIt and AlphaZero). In addition, unlike ExIt, after each match there is a learning phase: learning is carried out continuously. The acquired experience is thus immediately used to directly generate better matches.

Remark 27. Learning the minimax value (performed by Descent) instead of the end-game value (performed by AlphaZero and ExIt) is more informative (under the assumption that the state evaluation is of quality).

Remark 28. Note that in the case of our experiments with Athénan, we do not perform matches in parallel (it would however be possible to perform such parallelization). Conversely, ExIt and AlphaZero require massive parallelization of matches (which is done in the experiments in this paper). However, with Athénan, the evaluation of the child states of each state of the game tree is carried out in parallel (on a only GPU), which makes it possible to speed up searches without loss of efficiency. This lossless parallelization is not possible with ExIt and AlphaZero. More details on this parallelization, called Child Batching, are provided in (Cohen-Solal, 2025).

8.2.2 SUMMARY

In summary, the two points that seem most important are that, on the one hand, Athénan is based on Descent to generate matches, whereas AlphaZero uses MCTS and ExIt uses the neural network (without search). On the other, Athénan uses tree learning whereas AlphaZero and ExIt uses terminal learning and a target policy. In particular, Athénan does not use a policy, thus there is no need to encode actions, which poses a problem in certain contexts (Soemers, Mella, Browne, and Teytaud, 2022). Additionally, with Athénan, learning is done after each match. Finally, for confrontations, Athénan uses Unbounded Minimax with Safe Decision but AlphaZero and ExIt use a variant of MCTS biased to use their policy-value neural network.

8.3 The Experimental Comparison

We perform an experimental comparison between ExIt and Athénan. Specifically, for the training performed with Athénan, we use completed Descent (Algorithm 10) with tree learning (Algorithm 3), completed ordinal distribution (see Section 5 and Algorithm 14), the classic gain of the game as reinforcement heuristic[7] (see Section 6), and the stratified experience replay (Section 8).

We test these two algorithms on the following games: Go 9×9 , Go 11×11, Hex 11×11, Hex 13×13, Othello 8×8, Othello 10×10, Connect6, Outer-Open-Gomoku, Havannah 8, Havannah 10. We performed 40 repetitions per game and per algorithm. Each training process lasted 15 days.

8.4 Technical Details

We now present the technical details, starting with the parameters common to both approaches, then those specific to ExIt, and finally those specific to Athénan.

8.4.1 COMMON PARAMETERS

The training time for each neural network is 15 days.

We use the following neural network architecture for each of the training carried out (used as the adaptative evaluation function): a residual network with a convolutional layer with 132 filters, followed by 8 residual blocks (two 3×3 convolutions per block with 132 filters each), followed by a flat layer, and followed by two fully connected hidden layers (with each N neurons), followed by the final fully connected layer with 1 neuron (the output). The activation function used is the ReLU. The value N for each game is described in Table 5. The number of variables in each network is approximately $5 \cdot 10^6$. The Adam parameters are $\lambda = 0.0001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$. L_2 regularization is used with 0.001 as parameter.

Hardware: 8 cores 2.0 GHz processor (AMD Trento EPYC 7A53) with 32 Gio of DDR4-3200 MHz CPU memory, 1 Slingshot 200 Gb/s NICs, 1 GPUs devices (AMD MI250X accelerator) with a total of 64 Gio of HBM2 (MI250X partition of the Adastra supercomputer).

	N
Go 9×9	365
Go 11×11	228
Hex 11×11	228
Hex 13×13	155
Othello 8×8	477
Othello 10×10	286
Outer-Open-Gomoku	111
Connect6	65
Havannah 8	111
Havannah 10	65

Table 5: Number of hidden dense neurons N used by ExIt and Athénan for each studied game.

7. We could have used the score heuristic when available and the depth heuristic otherwise, which would have strongly increased Athénan's performance, but since it might be possible to inject such heuristics into ExIt, it is fair not to use them.

8.4.2 EXIT PARAMETERS

The neural network of ExIt has an additional head used to calculate the policy (parallel to the dense layers described above). It is composed of a convolutional 1×1 layer with 1 filter followed by a final dense layer with as many neurons as there are actions available (i.e. the number of neurons is the output policy size). Finally, the softmax function is applied on the output with the temperature parameter T_{softmax} (provided at the end of this section).

As described in the ExIt paper, the neural network is initialized using data from self-play matches of the base MCTS algorithm. In this paper, that data were generated and learned during 12 hours (10 MCTS has been used in parallel by using 10 CPU). From the third day, the value network is trained and used in addition to the policy network. We use the default settings of ExIt, except for the number of rollouts and the number of matches parallelization. We use 2500 rollouts for the MCTS algorithm used by ExIt instead of $10,000$. Recall that the parallelization of matches is used to perform matches synchronously so that their states are evaluated simultaneously by the neural network on the GPU. The inference batch size of this parallelization procedure has been reduced from 1024 to 500 (except for Othello 10×10 where it is reduced to 333). Based on our testings, these changes have very little impact on performance. However, the original larger values cause many memory overflows, some of which are unpredictable. Recall that ExIt has been applied in its introduction article only to Hex 9×9, which is a game with a quite small board.

For the other parameters, we thus use the default settings: UCT constant C during the initialization: 0.25, UCT constant C after the initialization: 0.05, learning batch size: 250, softmax temperature $T_{\text{softmax}} = 0.1$, search constants of ExIt MCTS: $w_a = 100$ and $w_v = 0.75$.

8.4.3 ATHÉNAN PARAMETERS

During training, search time per action is $\tau = 3s$. The stratified experience replay parameters used are: the batch size $B = 3000$, the memory size $\mu = 100$, the duplication factor $\delta = 3$. Moreover, when the children of a state are evaluated by the neural network, they are batched and thus evaluated in parallel (on the only GPU). The evaluation function has been pre-initialized by learning the values of random terminal states (of the order of $10,000,000$). This pre-initialized lasted 12 hours and used only one CPU. Resolved states are kept in memory (the memory of the resolved states is emptied every 12 learning hours). No simulated annealing was used for the ordinal distribution. The parameter value of this distribution is a random number drawn uniformly from [0,1] each time the ordinal distribution is used. Other parameters are the same as Section 3.3.1.

8.5 Results

Athénan and ExIt were evaluated against the base MCTS algorithm throughout their training. An evaluation has been performed approximately every day (for a total of 15 evaluations). The search time per action during the evaluations is 2 seconds. During an evaluation, each algorithm played 100 matches as first player and 100 more matches as second player.

The performance of an algorithm is its win rate minus its loss rate against MCTS, averaged over the 40 repetitions. The final performance improvement of Athénan compared to ExIt across all games is $+62.51\% \pm 6.3\%$ (p-value: 7.67×10^{-55}). The evolution during the learning process of the average performances over all games of Athénan and Exit against MCTS are described in Table 5. The final performance improvement of Athénan compared to ExIt and evolution curves detailed for each game are in Table 6.

In conclusion, Athénan performs better than ExIt on all games and is strongly better on average and on the majority of games.

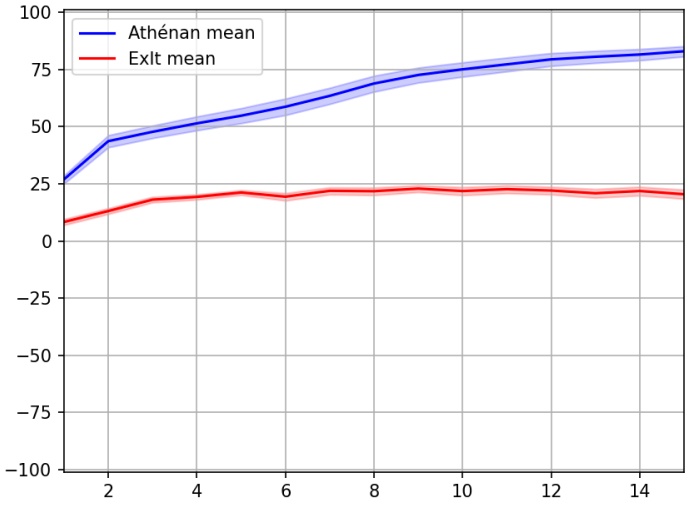

Figure 5: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS over all tested games (mean of performance and its stratified bootstrapping 95% confidence interval).

	mean	95% C.I.	repetitions	wilcoxon p-value	learning curve
Go 9×9	$+162\%$	$\pm 9.4\%$	40	1.82×10^{-12}	Table 9
Go 11×11	$+158\%$	$\pm 15.9\%$	40	1.27×10^{-11}	Table 10
Hex 11×11	$+32\%$	$\pm 10.9\%$	40	6.39×10^{-7}	Table 11
Hex 13×13	$+47\%$	$\pm 14.5\%$	40	2.37×10^{-5}	Table 12
Othello 8×8	$+110\%$	$\pm 5.4\%$	40	9.09×10^{-13}	Table 13
Othello 10×10	$+89\%$	$\pm 3.9\%$	40	9.09×10^{-13}	Table 14
Connect6	$+5.45\%$	$\pm 7.2\%$	40	1.88×10^{-5}	Table 16
Outer-Open-Gomoku	$+9.5\%$	$\pm 9.2\%$	40	6.39×10^{-7}	Table 15
Havannah 8	$+92\%$	$\pm 24.0\%$	40	9.09×10^{-13}	Table 17
Havannah 10	$+23\%$	$\pm 3.5\%$	40	4.33×10^{-8}	Table 18

Table 6: The final performance improvement and learning curves of Athénan compared to ExIt (each evaluation of each repetition is based on 200 matches ; C.I.: confidence interval).

9. Application to Hex

In this section, we apply Athénan to develop Hex program-players. We first present the game, including its rules and computational properties, followed by a review of existing Hex algorithms (Section 9.2). We then apply Athénan to boards of size 11×11 (Section 9.3) and 13×13 (Section 9.4), exceeding the level of 3HNN, the state-of-the-art, in both cases.

9.1 Game of Hex

The game of Hex (Browne, 2000) is a two-player combinatorial strategy game. It is played on an empty $n \times n$ hexagonal board. An $n \times n$ board is said to have size n. The board can be of any size, although the classic sizes are 11, 13 and 19. In turn, each player places a stone of his color on an empty cell (each stone is identical). The goal of the game is to be the first to connect the two opposite sides of the board corresponding to its color. Figure 6 illustrates an end game. Although its rules are simplistic, Hex tactics and strategies are complex. The number of states and the number of actions per state are very large, similar to the game of Go. For boards of size 11, the number of states exceeds that of chess (Table 6 of the work of Van Den Herik, Uiterwijk, and Van Rijswijck, 2002). For any board size, the first player has a winning strategy (Berlekamp, Conway, and Guy, 2003) which is unknown, except for board sizes smaller than or equal to 10 (Pawlewicz and Hayward, 2013) (the game is weaky solved up to the size 10). In fact, resolving a particular state is PSPACE-complete (Reisch, 1981; Bonnet, Jamain, and Saffidine, 2016). There is a variant of Hex using a swap rule. With this variant, the second player can play in first action a special action, called *swap*, which swaps the color of the two players (i.e. they swap their pieces and their sides). The swap rule reduces the imbalance between the two players (without the swap rule, the first player has a very strong advantage). It is generally used in competitions. It is always used at the Computer Olympiad and in this paper.

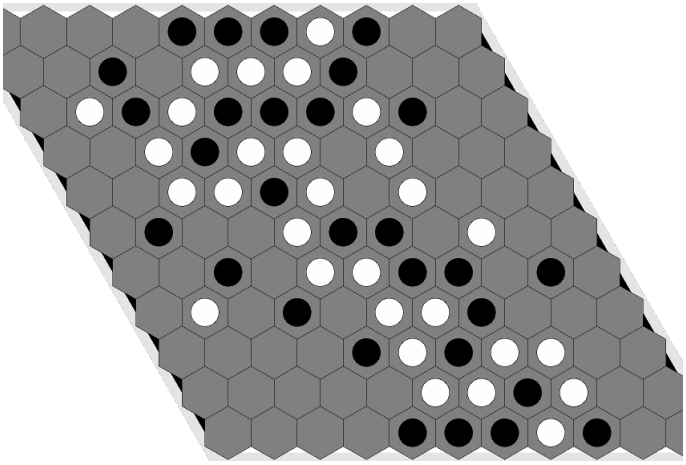

Figure 6: A Hex end game of size 11 (white wins)

9.2 Hex Programs

Many Hex player programs have been developed. For example, MoHex 1.0 (Huang, Arneson, Hayward, Müller, and Pawlewicz, 2013) is a program based on Monte Carlo tree search. It also uses many techniques dedicated to Hex, based on specific theoretical results. In particular, it is able to quickly determine a winning strategy for some states (without expanding the search tree) and to prune at each state many actions that it knows to be *inferior*. It also uses ad hoc knowledge to bias simulations of Monte Carlo tree search.

MoHex 2.0 (Huang et al., 2013) is an improvement of MoHex 1.0 that uses learned knowledge through supervised learning (namely correlations between victory and board patterns) to guide both tree exploration and simulations.

Other work then focused on predicting best actions, through supervised learning of a database of matches, using a neural network (Michalski, Carbonell, and Mitchell, 2013; LeCun, Bengio, and Hinton, 2015; Goodfellow, Bengio, Courville, and Bengio, 2016). The neural network is used to learn a *policy*, i.e. a prior probability distribution on the actions to play. These prior probabilities are used to guide the exploration of Monte Carlo tree search. First, there is MoHex-CNN (Gao et al., 2017) which is an improvement of MoHex 2.0 using a convolutional neural network (Krizhevsky et al., 2012). A new version of MoHex was then proposed: MoHex-3HNN (Gao et al., 2018). Unlike MoHex-CNN, MoHex-3HNN uses a residual neural network (He, Zhang, Ren, and Sun, 2016), which evaluates both policy, action, and state values. Furthermore, it does not perform simulations. Adding a value to actions allows MoHex-HNN to reduce the number of calls of the neural network, improving performance. MoHex-3HNN is the best Hex program (before Athénan). It won the Hex size 11 and 13 tournaments at the 2018 Computer Olympiad (Gao et al., 2019).

Programs which learn the evaluation function by reinforcement have also been designed. These programs are NeuroHex (Young et al., 2016), EZO-CNN (Takada, Iizuka, and Yamamoto, 2017), DeepEzo (Takada, Iizuka, and Yamamoto, 2019), and ExIt (Anthony et al., 2017). They learn from self-play. Unlike the other three programs, NeuroHex performs supervised learning (of a common Hex heuristic) followed by reinforcement learning. NeuroHex also starts its matches with a state from a database of games. EZO-CNN and DeepEzo use knowledge to learn winning strategies in some states. DeepEzo also uses knowledge during confrontations. ExIt learns a policy in addition to the value of states and it is based on MCTS (see Section 8 for details). It is the only program to have learned to play Hex without using knowledge. That result is, however, limited to the board size 9. A comparison of the main characteristics of these different programs is presented in Table 7.

9.3 A Long Training for Hex 11

We now apply all the techniques that we have proposed to carry out a long self-play reinforcement learning on Hex size 11. More precisely, we use completed Descent (Algorithm 10) with tree learning (Algorithm 3), completed ordinal distribution (see Section 5 and Algorithm 14), and the additive depth heuristic (see Section 6.2.2).

9.3.1 TECHNICAL DETAILS

In addition, we use a classical data augmentation: the adding of symmetrical states. Symmetrical states are added to D, the set of (s, v) pairs from the game tree to be learned (see Section 3 for details). The addition of symmetric states is performed after the end of each match and before the application of experience replay. Formally, $D \leftarrow D \cup \{(r_{180^\circ}(s), v) \mid (s, v) \in D\}$ where

$r_{180^\circ}(s)$ is s rotated by 180°. More precisely, the processing(D) method of Algorithm 3 is experience_replay(symmetry(D), μ, σ)) where symmetry(D) adds symetrical states in D as described above and returns D.

The used learning parameters are: search time per action $\tau = 2s$ and batch size $B = 3000$. The experience replay[8] parameters are: games memory size $\mu = 100$ and sampling rate $\sigma = 5\%$.

We use the following neural network as adaptative evaluation function: a residual network with a convolutional layer with 83 filters, followed by 4 residual blocks (2 convolutions per block with 83 filters each), and followed by a fully connected hidden layers (with 74 neurons). The activation function used is the ReLU. The Adam parameters are $\lambda = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$.

The input of the neural network is a game board extended by one line at the top, bottom, right and left (in the manner of the work of Young et al., 2016; Anthony et al., 2017). More precisely, each of these lines is completely filled with the stones of the player of the side where it is located. This extension is simply used to explicitly represent the sides of the board and their membership.

Moreover, when the children of a state are evaluated by the neural network, they are batched and thus evaluated in parallel (on the only GPU).

The evaluation function has been pre-initialized by learning the values of random terminal states (their number is $15,168,000$).

The other settings are the same as those in Section 3.3.1. Resolved states are kept in memory (the memory of the resolved states is emptied every two learning day).

The reinforcement learning process lasted $34,321$ matches. Note that the number of data used during the learning process is of the order of $59 \cdot 10^7$, the number of neural network evaluations is of the order of $196 \cdot 10^6$, and the number of state evaluations is of the order of $15 \cdot 10^9$.

Hardware: GPU Nvidia Tesla V100 SXM2 with 32 Go and 10 CPU (2,5 GHz ; processeurs Intel Xeon Gold 6248) with 40 Go RAM (gpu_p13 partition of Jean Zay supercomputer).

Remark 29. An alternative way to compare program strength in Hex is to iterate over all possible opening moves (play a match for each possible initial action). We do not adopt this approach here and instead use a standard evaluation protocol. These two evaluations measure different aspects: the standard evaluation assesses how strong one algorithm is relative to another in realistic play,

8. A variant of replay experience was applied, it memorizes the pairs of the last 100 games and not the last 100 pairs.

Programs	Size	Search	Learning	Network	Use
MoHex-CNN	13	MCTS	supervised	convolutional	policy
MoHex-3HNN	13	MCTS	supervised	residual	policy, state, action
NeuroHex	13	none	supervised, reinforcement	convolutional	state
EZO-CNN	7, 9, 11	Minimax	reinforcement	convolutional	state
DeepEZO	13	Minimax	reinforcement	convolutional	policy, state
ExIt	9	MCTS	reinforcement	convolutional	policy, state

Table 7: Comparison of the main features of the latest Hex programs. These characteristics are respectively the board sizes on which learning is based, the used tree search algorithm, the type of learning, the type of neural network, and its use (to approximate the values of states, actions, and/or policy).

whereas the alternative better captures the understanding of game strategies by evaluating performance over a set of alternative games in which the first move is fixed. However, the alternative thus evaluates performance over matches that may never occur in practice. This alternative setting can be useful to assess generality (e.g., from a learning perspective), but it may also bias the evaluation when considering algorithms specifically optimized for strong play in practically relevant positions. Note that if one aims to perform well under this alternative protocol, it is sufficient to include such opening-move iteration during training as well.

9.3.2 RESULTS

Table 8 shows Athénan's winning percentages against MoHex 3HNN using the evaluation function learned by Descent. Athénan surpasses the performance of MoHex 3HNN on 11×11 Hex without relying on prior knowledge.

search time	MoHex 2nd	MoHex 1st	**mean**	95% C.I.	matches	binomial p-value
$2.5s$	98%	84%	91%	2%	1000	1.12×10^{-171}
$10s$	91%	86%	88%	2%	1600	7.38×10^{-229}

Table 8: Winning percentages against MoHex 3HNN of $\mathrm{UBFM_s}$ using the learned evaluation function of Section 9.3 (the search time per action is the same for each player ; default settings are used for MoHex ; there are as many matches in first as in second player ; C.I.: confidence interval).

9.4 A Long Training for Hex 13

We carry out the same experiment as the previous section, but on Hex size 13.

9.4.1 TECHNICAL DIFFERENCES

We use the same parameters as for training on 11×11 Hex, except for the following parameters.

The architecture of the neural network is a convolutional layer with 186 filters, followed by 8 residual blocks (2 convolutions per block with 186 filters each), and followed by 2 fully connected hidden layers (with 220 neurons each). The activation function used is the ReLU. The network was not initialized using random end state values. The modified[9] experience replay parameters are: games memory size $\mu = 250$ and sampling rate $\sigma = 2\%$. The search time per action τ is $5s$. The reinforcement learning process lasted $5,288$ matches. The number of data used during the learning process is of the order of $16 \cdot 10^7$. The number of neural network evaluations is of the order of $64 \cdot 10^6$. The number of state evaluations is of the order of $7 \cdot 10^9$.

9.4.2 RESULTS

Table 9 shows Athénan's winning percentages against MoHex 3HNN on size 13 boards, also surpassing 3HNN without using prior Hex knowledge.

9. A variant of experience replay was applied: it memorizes the pairs from the last 250 games rather than the last 250 pairs.

10. Application to Othello

In this section, we apply Athénan to the game of Othello. After completing the reinforcement learning process without prior knowledge, we evaluated Athénan against the state-of-the-art Othello program, Edax[10]. We start by presenting the state-of-the-art at Othello (Section 10.1), then we describe the experiment carried out (Section 10.2) and its technical details (Section 10.3), and finally we present the results (Section 10.4).

10.1 Othello Related Work

Although Othello was one of the first games in which artificial intelligence achieved superhuman performance, research on Othello continues. Edax and Saio are the Othello state-of-the-art programs (Wikipedia contributors, 2022; Norelli and Panconesi, 2022; Liskowski, Jaśkowski, and Krawiec, 2018; Fédération Française d'Othello). Saio is not free and is similar to Edax. Edax is an open source highly optimized program. It is based on minimax and dedicated Othello techniques: *multi-probcut tree search*, *tabular value functions* (as evaluation function) generated from expert knowledge, and opening sequences (Buro, 1997).

A move predictor (Liskowski et al., 2018) was also designed at Othello using a convolutional neural network learned by supervised learning. However, its level is low. In particular, it only surpasses Edax when Edax plans at depth 2 (which takes it on the order of 10^{-4} seconds), whereas Edax plans with a depth between 20 and 30 for a search time of the order of seconds.

In addition, AlphaZero has been applied recently at Othello (Norelli and Panconesi, 2022). This AlphaZero program, named Olivaw, is as strong as Edax with a search depth of 8. It reached a high level in Othello and was able to surpass a national champion.

A detailed discussion of related work on computer Othello is provided by Norelli and Panconesi (2022); Wikipedia contributors (2022); Liskowski et al. (2018).

10.2 Experiment

We now apply Athénan to carry out a long self-play reinforcement learning at Othello. More precisely, we use completed Descent (Algorithm 10) with tree learning (Algorithm 3), completed ordinal distribution (see Section 5 and Algorithm 14), the game score as reinforcement heuristic (see Section 6), and the stratified experience replay (Section 8).

10. Edax version 4.4: `https://github.com/abulmo/edax-reversi`

search	MoHex 2nd	MoHex 1st	**mean**	95% C.I.	total matches	binomial p-value
$2.5s$	100%	100%	100%	0%	1200	0
$10s$	100%	100%	100%	0%	800	1.50×10^{-241}

Table 9: Winning percentages against MoHex 3HNN of $\mathrm{UBFM_s}$ using the learned evaluation function of Section 9.4 (the search time per action is the same for each player ; default settings are used for MoHex ; there are as many matches in first as in second player ; C.I.: confidence interval).

10.3 Technical Details

In addition, we use a classical data augmentation: the adding of symmetrical states. Symmetrical states are added in D, the set of (s, v) pairs from the game tree to be learned (see Section 3 for details). This addition is performed after the end of each match and before the application of experience replay. Formally, $D \leftarrow D \cup \{(\mathrm{sym}(s), v) \mid (s, v) \in D\}$ where $\mathrm{sym}(s)$ randomly returns one of the 8 symmetric board states of s. In other words, the processing(D) method of Algorithm 3 is experience_replay(symmetry(D), μ, σ)) where symmetry(D) adds symetrical states in D as described above and returns D.

Search time per action is $\tau = 5s$. The stratified experience replay parameters used are: the batch size $B = 4000$, the memory size $\mu = 100$, the duplication factor $\delta = 3$.

We use the following neural network as adaptative evaluation function: a residual network with a convolutional layer with 373 filters, followed by 8 residual blocks (two 3×3 convolutions per block with 373 filters each following with a squeeze-and-excitation layer (Hu, Shen, and Sun, 2018) whose ratio is 16), followed by a 1×1 convolution with 4453 filters, followed by a global sum pooling (Aich and Stavness, 2018), followed by a flat layer, and followed by a fully connected hidden layers (with 4453 neurons). The activation function used is the ReLU. Number of weights of the neural network is $41,695,147$. The Adam parameters are $\lambda = 0.0001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-5}$. The parameter of the L_2 regularization is 0.99.

Moreover, when the children of a state are evaluated by the neural network, they are batched and evaluated in parallel on the single GPU. The evaluation function has been pre-initialized by learning the values of random terminal states (of the order of $10,000,000$). Resolved states are kept in memory (the memory of the resolved states is emptied every 12 learning hours). No simulated annealing was used for the ordinal distribution. The parameter of the ordinal distribution is a random number drawn uniformly from [0,1] each time this distribution is used. The other settings are the same as those in Section 3.3.1.

The reinforcement learning process lasted $52,470$ matches. The number of data used during the learning process is of the order of $426 \cdot 10^7$. The number of neural network evaluations is of the order of $606 \cdot 10^6$, and the number of state evaluations is of the order of $3.49 \cdot 10^9$.

Hardware: GPU Nvidia Tesla V100 SXM2 with 32 Go and 10 CPU (2,5 GHz ; processeurs Intel Xeon Gold 6248) with 40 Go RAM (gpu_p13 partition of Jean Zay supercomputer).

10.4 Results

The winning percentages of Athénan (i.e., of $\mathrm{UBFM_s}$ using the learned evaluation function generated by Descent) against Edax are presented in Table 10. These results demonstrate that the proposed techniques surpass the level of Edax at Othello. Recall that this reinforcement learning was carried out without using prior knowledge about Othello.

11. Application to Arimaa

In this section, we apply Athénan to the game of Arimaa and evaluate the reinforcement learning process without prior knowledge against the state-of-the-art program, Sharp (Wu, 2015). We start by presenting the state-of-the-art at Arimaa (Section 11.1), then we describe the experiment carried out (Section 11.2) and its technical details (Section 11.4), and finally we present the results (Section 11.5).

11.1 Arimaa Related Work

Since 2015, the state-of-the-art in Arimaa has been the program Sharp (Wu, 2015). Sharp combined traditional alpha–beta pruning with handcrafted heuristic functions. It also incorporates numerous alpha–beta enhancements, such as killer moves, history heuristic, quiescence search, extensions, and late move reduction, all of which require dedicated knowledge to implement. It also uses techniques dedicated to Arimaa: specific algorithms and knowledge used by alpha-beta and its improvements (mainly for move ordering, pruning, and state evaluating).

Alternative approaches to alpha-beta at Arimaa have been studied, notably based on Monte Carlo, but without success (Wu, 2015). There have also been some attempts to apply AlphaZero to Arimaa, but to our knowledge, they have all failed (Forum, 2024).

11.2 Experiment

We now apply Athénan to carry out a long self-play reinforcement learning at Arimaa. More precisely, we use completed Descent (Algorithm 10) with tree learning (Algorithm 3), completed ordinal distribution (see Section 5 and Algorithm 14), the stratified experience replay (Section 8), and the Arimaa reinforcement heuristic described in the following section.

11.3 Arimaa Reinforcement Heuristic

The reinforcement heuristic used for Arimaa is a variation of the presence heuristic that accounts for the importance of pieces according to the game rules (Section 6).

We use the following state evaluation function, denoted by $\mathrm{h}_{\mathrm{arimaa}}$, as reinforcement heuristic. The function $\mathrm{h}_{\mathrm{arimaa}}$ is defined by $\mathrm{h}_{\mathrm{arimaa}}(s) = \begin{cases} 69 + V(s) & \text{if first player wins} \\ -69 + V(s) & \text{if second player wins} \end{cases}$ where s is a game state, $P_1(s)$ (resp. $P_2(s)$) is the set of pieces of the first player (resp. second player) in s, $V(s)$ is the difference of the values of the two players pieces:

$$V(s) = \sum_{p \in P_1(s)} v_{\mathrm{arimaa}}(p) - \sum_{p \in P_2(s)} v_{\mathrm{arimaa}}(p),$$

and $v_{\mathrm{arimaa}}(p)$ is the value the piece p corresponding to the importance of that piece according to the rule. The value $v_{\mathrm{arimaa}}(p)$ is defined as follows: $v_{\mathrm{arimaa}}(\mathrm{elephant}) = 5$, $v_{\mathrm{arimaa}}(\mathrm{camel}) = 4$, $v_{\mathrm{arimaa}}(\mathrm{horse}) = 3$, $v_{\mathrm{arimaa}}(\mathrm{dog}) = 2$, $v_{\mathrm{arimaa}}(\mathrm{cat}) = 1$, $v_{\mathrm{arimaa}}(\mathrm{rabbit}) = 6$.

search time	win - loss	win	draw	loss	95% C.I.	matches	p-value
$1.5s$	20%	39.37%	41.25%	19.37%	$3,4\%$	800	8.76×10^{-14}
$15s$	12.25%	25.37%	61.5%	13.12%	$3,4\%$	800	1.39×10^{-25}

Table 10: Winning percentages of $\mathrm{UBFM_s}$ against Edax using the learned evaluation function of Section 10.2 (the search time per action is the same for each player ; there are as many matches in first as in second player ; note: the binomial p-value is without draw matches ; C.I.: confidence interval).

11.4 Technical Details

In addition, we use a classical data augmentation: the adding of symmetrical states. Symmetrical states are added in D, the set of (s, v) pairs from the game tree to be learned (see Section 3 for details). This addition is performed after the end of each match and before the application of experience replay. Formally, $D \leftarrow D \cup \{(\mathrm{sym}(s), v) \mid (s, v) \in D\}$ where $\mathrm{sym}(s)$ returns the symmetric board state of s. In other words, the processing(D) method of Algorithm 3 is experience_replay(symmetry(D), μ, σ)) where symmetry(D) adds symetrical states in D as described above and returns D.

Search time per action is $\tau = 5s$. The stratified experience replay parameters used are: the batch size $B = 4000$, the memory size $\mu = 100$, the duplication factor $\delta = 3$.

During training, we applied an additional game rule to avoid overly long games: if the number of actions played exceeds 900, the match is declared a draw.

We use the following neural network as adaptative evaluation function: a residual network with a convolutional layer with 263 filters, followed by 4 residual blocks (two 3×3 convolutions per block with 263 filters each), followed by a 1×1 convolution with 2217 filters, followed by a global sum pooling (Aich and Stavness, 2018), followed by a flat layer, and followed by a fully connected hidden layers (with 2217 neurons). The activation function used is the ReLU. Number of weights of the neural network is $10,511,017$. The Adam parameters are $\lambda = 0.0001$, $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\epsilon = 10^{-7}$.

Moreover, when the children of a state are evaluated by the neural network, they are batched and thus evaluated in parallel (on the only GPU). The evaluation function has been pre-initialized by learning the values of random terminal states (of the order of $10,000,000$). Resolved states are kept in memory (the memory of the resolved states is emptied every 12 learning hours). The other settings are the same as those in Section 3.3.1.

The reinforcement learning process lasted $6,317$ matches. The number of data used during the learning process is of the order of $768 \cdot 10^7$. The number of neural network evaluations is of the order of 10^9. The number of state evaluations is of the order of $30 \cdot 10^9$.

Hardware: GPU Nvidia Tesla V100 SXM2 with 32 Go and 10 CPU (2,5 GHz ; processeurs Intel Xeon Gold 6248) with 40 Go RAM (gpu_p13 partition of Jean Zay supercomputer).

11.5 Results

The winning percentages of Athénan (i.e., $\mathrm{UBFM_s}$ using the learned evaluation function generated by Descent) against Sharp are shown in Table 11. These results demonstrate that the proposed techniques surpass the level of Sharp in Arimaa. Recall that this reinforcement learning process was carried out without using prior knowledge about Arimaa strategies.

search time	win	loss	95% C.I.	total matches	binomial p-value
$2.5s$	90.33%	9.66%	4%	600	8.32×10^{-99}
$10s$	92.5%	7.5%	4%	600	4.19×10^{-113}

Table 11: Winning percentages against Sharp of $\mathrm{UBFM_s}$ using the learned evaluation function of Section 10.2 (the search time per action is the same for each player ; there are as many matches in first as in second player ; C.I.: confidence interval).

12. Application to Morpion Solitaire

In this section, we apply Athénan to *Morpion Solitaire* (also called *Join Five*), a single-player puzzle game. We start by detailing the rules of Morpion Solitaire (Section 12.1), then presenting the state-of-the-art at Morpion Solitaire (Section 12.2). Next, we describe the experiment carried out (Section 12.3) and its technical details (Section 12.4), and finally we present our results (Section 12.5).

12.1 Morpion Solitaire Rules

The goal of the Morpion Solitaire is to play as long as possible. The score of the Morpion Solitaire is therefore the number of moves played since the start of the game. The Morpion Solitaire board is a square grid of theoretically infinite size. At the start of the game, some points are already placed in the shape of a cross. At each turn, the player places a point on the board and crosses out an alignment of 5 points not already crossed out in that direction. A point can therefore only be crossed out at most 4 times (horizontally, vertically, diagonally and antidiagonally). The 5 points must include the point just placed. A point can only be placed if it will belong to an alignment of 5 points that can be crossed out. A move includes the point placement plus the choice of the 5 aligned points and their crossing-out. As soon as the player cannot place any more points, the game is over. The initial board and the board of an example of first move are represented in Figure 7.

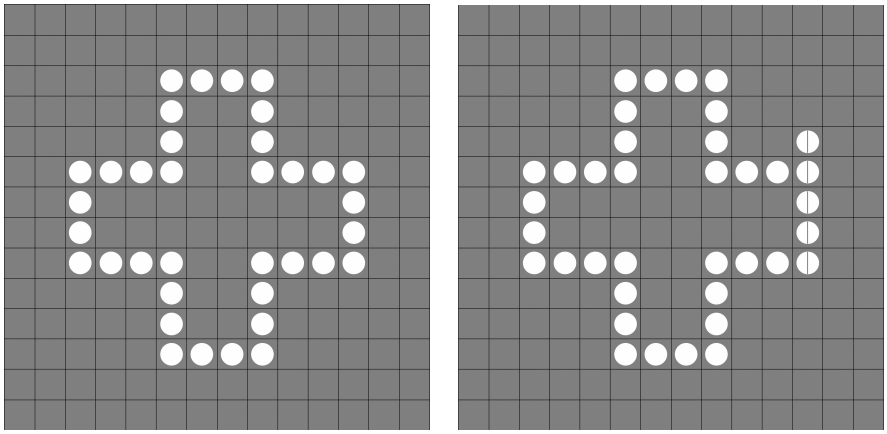

Figure 7: Initial board for Morpion Solitaire (left) ; The board of Morpion Solitaire after a first move (right).

Morpion Solitaire has a variant called the *touching version*; the original version described above is referred to as the *disjoint version*. With the touching variant, the endpoints of an already drawn line can be used as an endpoint for a new line aligned in the same direction. This variant is not studied in this paper.

12.2 Morpion Solitaire Related Work

An upper bound on the maximum score at Morpion Solitaire (disjoint version) is 84 (Michalewski, Nagórko, and Pawlewicz, 2016). The human record is 68 (Wang, Preuss, Emmerich, and Plaat,

2020). The best actual record is 82. The record of 82 was obtained with *Nested Rollout Policy Adaptation* (Rosin, 2011) and *Beam Nested Rollout Policy Adaptation* (Cazenave and Teytaud, 2012), which are Monte Carlo algorithms, using online learning, dedicated to one-player games.

AlphaZero has been applied with the *Ranked Reward* technique to Morpion Solitaire (Wang et al., 2020). The record obtained by this AlphaZero program is only 67.

ExIt, the reinforcement learning algorithm without knowledge, has been combined with Nested Rollout Policy Adaptation and applied to Morpion Solitaire (Doux, Negrevergne, and Cazenave, 2021). The record of this ExIt program is 73.

The touching version has also been studied. The record for this variant of Morpion Solitaire is 178 (Nagórko, 2019).

12.3 Experiment

We now apply Athénan to carry out a long self-play reinforcement learning without knowledge at Morpion Solitaire. More precisely, we use Descent (Algorithm 9) with tree learning (Algorithm 3), ordinal distribution (see Section 7), the stratified experience replay (Section 8), and the score heuristic as reinforcement heuristic (Section 6).

Remark 30. Note that completion is not used since it only concerns two-player games. However, resolved states are still used. In this context, a state is resolved only if all its children are resolved.

12.4 Technical Details

All endgames are draws unless a score of 84 is reached, which counts as a win.

We use a classical data augmentation: the adding of symmetrical states. Symmetrical states are added in D, the set of (s, v) pairs from the game tree to be learned (see Section 3 for details). This addition is performed after the end of each match and before the application of experience replay. Formally, $D \leftarrow D \cup \{(\mathrm{sym}\,(s)\,, v) \mid (s, v) \in D\}$ where $\mathrm{sym}\,(s)$ randomly returns one of the symmetric board states of s. In other words, the processing(D) method of Algorithm 3 is experience_replay(symmetry(D), μ, σ)) where symmetry(D) adds symetrical states in D as described above and returns D.

Search time per action is $\tau = 1s$. The stratified experience replay parameters used are: the batch size $B = 3000$, the memory size $\mu = 100$, the duplication factor $\delta = 3$.

We use the following neural network as adaptative evaluation function: a residual network with a convolutional layer with 186 filters, followed by 8 residual blocks (two 3×3 convolutions per block with 186 filters each, following with a squeeze-and-excitation layer (Hu et al., 2018) whose ratio is 16), followed by a flat layer, and followed by two fully connected hidden layers (with each 104 neurons). The activation function used is the ReLU. Number of weights of the neural network is $9,989,285$. The Adam parameters are $\lambda = 0.0001$, $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\epsilon = 10^{-8}$.

Moreover, when the children of a state are evaluated by the neural network, they are batched and thus evaluated in parallel (on the only GPU). The evaluation function has been pre-initialized by learning the values of random terminal states (of the order of $10,000,000$). Resolved states are kept in memory (the memory of the resolved states is emptied every 12 learning hours). The other settings are the same as those in Section 3.3.1.

The reinforcement learning process lasted $58,971$ matches. The number of data used during the learning process is of the order of $126 \cdot 10^7$. The number of neural network evaluations is of the order of $0.19 \cdot 10^9$. The number of state evaluations is of the order of $1.8 \cdot 10^9$.

Hardware: GPU Nvidia Tesla V100 SXM2 with 32 Go and 10 CPU (2,5 GHz ; processeurs Intel Xeon Gold 6248) with 40 Go RAM (gpu_p13 partition of Jean Zay supercomputer).

12.5 Results

Figure 8 shows the score progression throughout training for Morpion Solitaire using Athénan (i.e., $\mathrm{UBFM_s}$ with the evaluation function generated by Descent) with 0.1 seconds of search per move. The score of 82 is reached in the $58,971$ match.

Thus, Athénan reaches the state-of-the-art level using a more general algorithm than previous specialized techniques for this game. Recall that this result was not achieved with ExIt and AlphaZero, the two alternative algorithms of Athénan.

This result further shows that Athénan is applicable to single-player games.

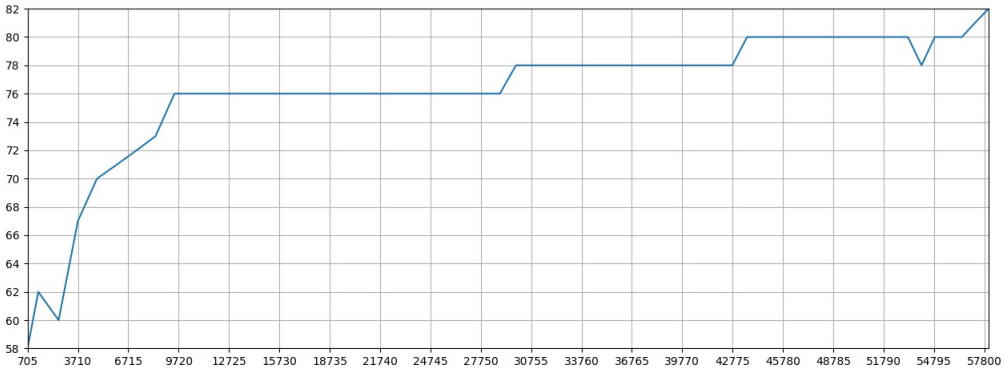

Figure 8: Evolution during training of the score obtained in Morpion Solitaire using the evaluation function learned and $\mathrm{UBFM_s}$ with 0.1 second of search per action (the x-axis shows the number of training matches) .

13. Computer Olympiad Results

In this section, we briefly present Athénan's results in the global board game artificial intelligence competition, namely the Computer Olympiad.

Athénan has won a total of 57 gold medals at the Computer Olympiad. It won nine gold medals in 2025, eleven gold medals in 2024 (Cohen-Solal and Cazenave, 2025a), sixteen gold medals in 2023 (Cohen-Solal and Cazenave, 2023a), five gold medals in 2022, eleven in 2021, and again five in 2020 (Cohen-Solal and Cazenave, 2021).

Moreover, Athénan is currently the defending champion on 20 games (the nine 2025 gold medals plus eleven previous uncontested gold medals). All Athénan results are summarized in Table 12.

No other program has won more than five gold medals in a single year since the competition began in 1989. Achieving five gold medals in a single year was an exceptional accomplishment.

11. Participation in the competition using the stochastic generalization of Athénan (Cohen-Solal and Cazenave, 2023b)

14. Conclusion

This section concludes the paper by summarizing the Athénan framework, reviewing the performance evaluations, and outlining perspectives for future work.

14.1 New algorithms: Athénan components

We proposed several new techniques for reinforcement learning state evaluation functions.

Firstly, we have generalized tree bootstrapping (tree learning) in the context of reinforcement learning without knowledge based on non-linear functions. We showed that learning the values of the partial game tree, rather than only the root, significantly improves learning performance.

Secondly, we have introduced the Descent Minimax algorithm, which explores in a manner similar to Unbounded Minimax and is intended for use during learning. Unlike Unbounded Minimax, Descent iteratively explores sequences of best actions *until terminal states*. Its objective is to improve the quality of data used during learning while retaining the advantages of Unbounded

	2020	2021	2022	2023	2024	2025
Amazons	Gold	Gold	1	Gold	Gold	1
Breakthrough	Gold	Gold	Gold	Gold	Gold	Gold
Clobber	Gold	1	1	Gold	Gold	Gold
Surakarta	Gold	Gold	Gold	Gold	Gold	Gold
Othello 8x8	Silver	Gold	Silver	Bronze	1	1
Othello 10x10	Gold	1	1	Gold	1	1
Othello 16x16					Gold	1
Hex 11x11		Gold	1	Gold	Gold	Gold
Hex 13x13	0	Gold	1	Gold	Gold	Gold
Hex 19x19	0	Gold	1	Gold	1	1
Havannah 8	0	Gold	1	Gold	Gold	1
Havannah 10		Gold	1	Gold	1	1
Canadian Draughts		Gold	Gold	Gold	1	1
Brazilian Draughts		Gold	0	Silver	1	1
International Draughts		Silver	0	Silver	0	Gold
Connect6		Silver	Silver	Silver	0	0
Outer-Open-Gomoku		Bronze	Silver	Bronze	0	Silver
Ataxx			Gold	Gold	Gold	1
Santorini			Gold	Gold	0	Gold
Lines of Action			0	Gold	Gold	Gold
Xiangqi			0	Gold	1	1
Arimaa				Gold	1	1
Shobu					Gold	1
Backgammon[11]						Gold

Table 12: Athénan Computer Olympiad results (0: participation without result ; 1: participation without opponent, i.e. title not contested ; empty: no participation).

Minimax. In our experiments, Descent outperforms Alpha-Beta, Unbounded Minimax, and MCTS by a significant margin.

Thirdly, we proposed the completion technique which allows to take into account the resolution of states, notably in the context of Unbounded Minimax and Descent. Note that the impact assessment of this technique was carried out in (Cohen-Solal and Cazenave, 2025b). This technique improves performance but the gain is lower than the techniques evaluated in this article.

Fourthly, we have suggested to replace the classic gain of a game ($+1/0/-1$) by different terminal evaluation functions, called reinforcement heuristics. We proposed several general terminal evaluation functions, such as the depth heuristic, which accounts for game duration to favor quick wins and slow defeats. Our experiments have shown that the use of a reinforcement heuristic improves performances. Our study recommends using the score heuristic when the game has one and otherwise using the depth heuristic.

Fifth, we have proposed a new action selection distribution, called ordinal distribution, which does not take into account the value of states but only their order.

We combined all these techniques in Athénan, a zero-knowledge reinforcement learning algorithm, which is thus an alternative to AlphaZero or ExIt.

14.2 Evaluation of Athénan

In the unpublished initial version of this article (Cohen-Solal, 2020), Athénan was not compared to the two state-of-the-art zero-knowledge reinforcement learning algorithms, AlphaZero and ExIt. Meanwhile, Athénan's comparison with AlphaZero was conducted in collaboration with Tristan Cazenave in a separate study (Cohen-Solal and Cazenave, 2023c). In that study, Athénan using the classic game gain as reinforcement heuristic was at least seven times faster than AlphaZero on equivalent hardware. It is also at least 30 times faster using the reinforcement heuristics presented in this article. Athénan with one GPU even outperforms AlphaZero with 100 GPUs on some games.

In the new version of this paper, we have therefore added the missing experience: the comparison with ExIt. In the context of the experiments conducted in this article, Athénan performs much better than ExIt. ExIt's average performance does not even reach in 15 days the performance that Athénan obtained in one day. Note that in this study, Athénan only used the classic game gain as reinforcement heuristic. Thus, we should achieve a result 4 times higher with the advanced reinforcement heuristics proposed in this article.

We also showed that Athénan, without using any dedicated knowledge[12], surpassed the state-of-the-art in several games, including Hex (sizes 11 and 13), Othello, and Arimaa. This contrasts with the outperformed state-of-the-art programs which use dedicated knowledge.

Moreover, Athénan reached the top performance in Morpion Solitaire. The record for this game was previously achieved only by algorithms restricted to single-player settings. This result demonstrates that Athénan, in addition to two-player games, can also effectively handle single-player games. This is all the more noteworthy given that attempts to reach state-of-the-art performance in Morpion Solitaire using ExIt and AlphaZero have failed.

Finally, we summarized Athénan's results at the Computer Olympiad, the main international competition for board game AI. Athénan broke all records at this competition, tripling the record of

12. When we refer to *without knowledge*, we mean without any knowledge beyond the rules of the game or trivial consequences thereof (e.g., board symmetry, edge membership encoding in Hex, or piece importance order in Arimaa).

gold medals obtained in a single year (reaching 16 gold medals), winning a total of 57 gold medals in 6 years since its first participation, and being the defending champion on 20 games.

Overall, the results indicate that Athénan and its components provide efficient and effective alternatives to approaches based on Alpha-Beta and MCTS in the context of modern reinforcement learning.

Remark 31. Athénan has been programmed in Python. Switching to a faster programming language should reduce the learning time by a factor between two and five and will increase the winning percentages during confrontations.

14.3 Perspectives

We did not evaluate the ordinal distribution in this article. In a future work, we will show that this technique improves average performance. However, its impact is substantially lower than that of the other techniques proposed in this article (except completion).

As suggested by recent advancements in the field, evaluating the learned value functions by their ability to accelerate perfect solvers (e.g., reducing node counts in solving 11×11 Hex) would provide an alternative measure of knowledge quality in this context, less sensitive to testing-time hyperparameters.

A theoretical analysis of generalized tree bootstrapping with non-linear approximations would be valuable to formally establish convergence behaviors and stability bounds in knowledge-free reinforcement learning.

Ongoing work focuses on generalizing and adapting these algorithms to several contexts: stochastic games (Cohen-Solal and Cazenave, 2023b), multiplayer games (Cohen-Solal, 2026c, 2021), policy usage, and designing algorithms capable of challenging without overwhelming an opponent without prior knowledge of the adversary's game level (Cohen-Solal, 2026b).

A promising research perspective is the parallelization of Unbounded Minimax and Descent when learning: Unbounded Minimax determines the next action to play and Descent determines the pairs to learn (by batching evaluations on a single GPU or using two GPUs).

Another promising perspective is to train Athénan using two neural networks playing against each other. This approach could potentially reduce numerical issues and the risk of local optima. To maintain learning speed, each player performs a Descent search during the opponent's turn, while the neural networks are evaluated in parallel, either on the same GPU or on two separate GPUs. This approach can also be generalized to a pool of neural networks performing matches in parallel. In this context, it might be more interesting to replace the ordinal distribution with the max distribution. These approaches should speed up training and increase efficiency, but they will reduce effectiveness.

The other research perspectives include the application of our contributions to General Game Playing (at first with perfect information). A yet-to-be-determined Athénan parameterization, potentially using small neural networks, could achieve competitive performance in the context of General Game Playing. A modification of Athénan that could be interesting in this context is to perform "online" learning, i.e. instead of performing the learning phase after each match, to perform it after each search, or even after each Descent iteration, or even after each update of the game tree nodes.

The other research perspectives also include the application and adaptation of our contributions to the contexts of hidden information.

Finally, these perspectives include applying our contributions to optimization problems, such as the RNA Inverse Folding problem (Cazenave and Fournier, 2020; Cazenave and Touzani, 2024) and to the design of new algorithms, similar to AlphaZero's applications to matrix multiplication and sorting (Mankowitz, Michi, Zhernov, Gelmi, Selvi, Paduraru, Leurent, Iqbal, Lespiau, Ahern, et al., 2023; Fawzi, Balog, Huang, Hubert, Romera-Paredes, Barekatain, Novikov, R Ruiz, Schrittwieser, Swirszcz, et al., 2022). There are many other possible applications. We obtained promising results regarding the application of Athénan in the context of computer drawing (it is able to reproduce an image, with slight noise, without copying it, drawing it pixel by pixel) and in the context of cryptanalysis (chosen-plaintext attack with small-size keys). Finally, applying Athénan should improve the performance of generative AI systems.

Acknowledgments

This work was supported in part by the French government under management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, benchmark ANR19-P3IA-0001 (PRAIRIE 3IA Institute).

This project was provided with computer and storage resources by GENCI at CINES and IDRIS thanks to the grants 2020-AD010114027, 2020-AD011011772, 2020-AD011011714, 2021-AD011011461R1, 2022-AD011011461R2, 2023-AD011011461R3, 2023-A0141011461, 2024 - A0161011461, and 2025-A0181011461 on the supercomputers Jean Zay and Adastra (V100 and MI2050x partitions).

I thank Eric Piette for his valuable work on Ludii.

I thank GREYC for giving me access to its computation server, which allowed me to perform several experiments that inspired those in this article.

Finally, I would like to express my sincere gratitude to Tristan Cazenave, without whom I would not have had the financial and material resources to complete this study.

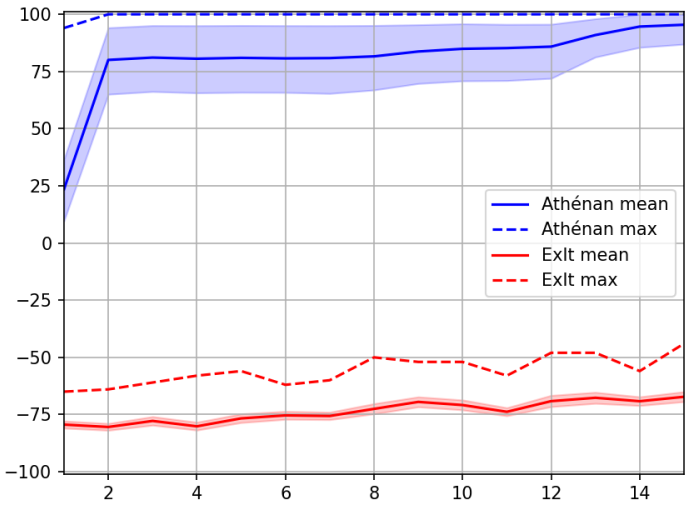

Figure 9: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Go 9×9 (maximum performance, and mean performance with its bootstrap 95% confidence interval).

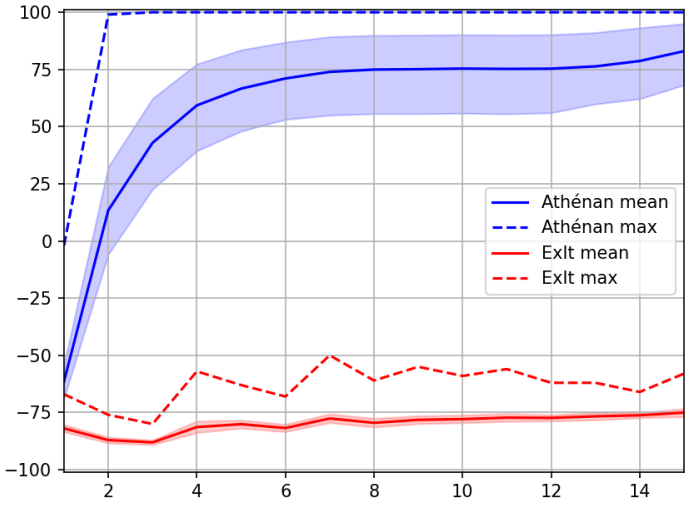

Figure 10: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Go 11×11 (maximum performance, and mean performance with its bootstrap 95% confidence interval).

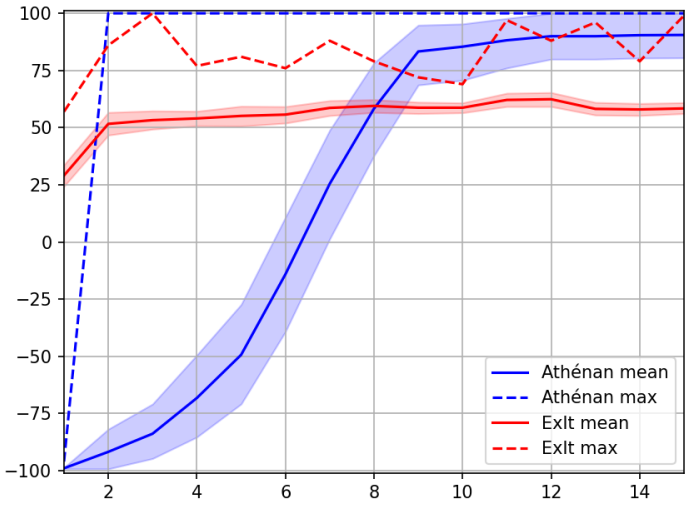

Figure 11: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Hex 11×11 (maximum performance, and mean performance with its bootstrap 95% confidence interval).

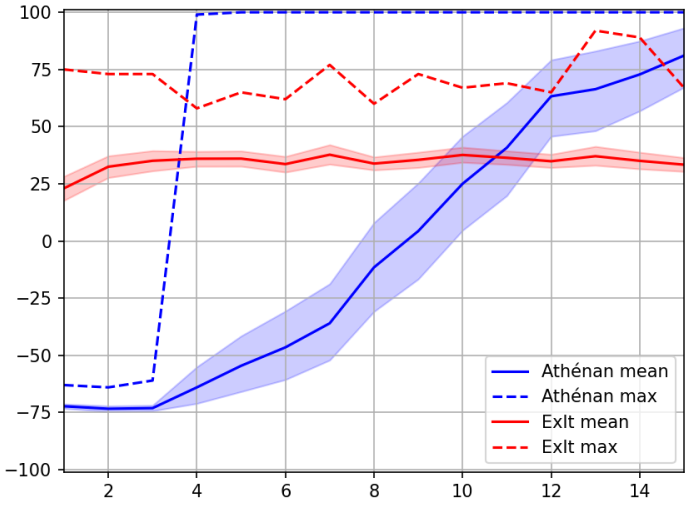

Figure 12: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Hex 13×13 (maximum performance, and mean performance with its bootstrap 95% confidence interval).

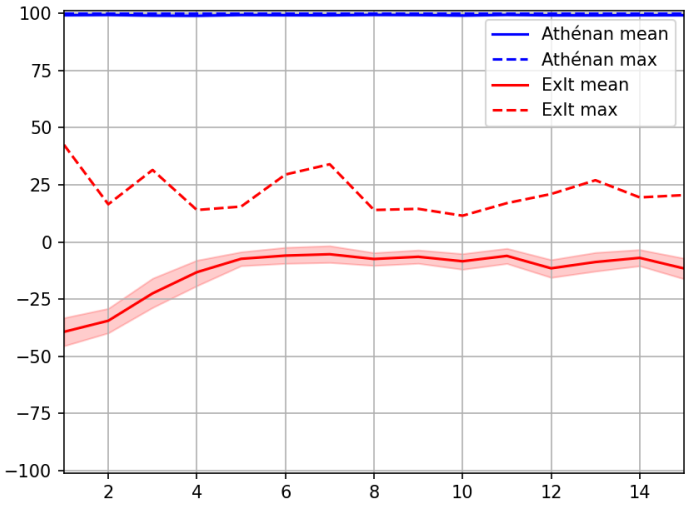

Figure 13: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Othello 8×8 (maximum performance, and mean performance with its bootstrap 95% confidence interval).

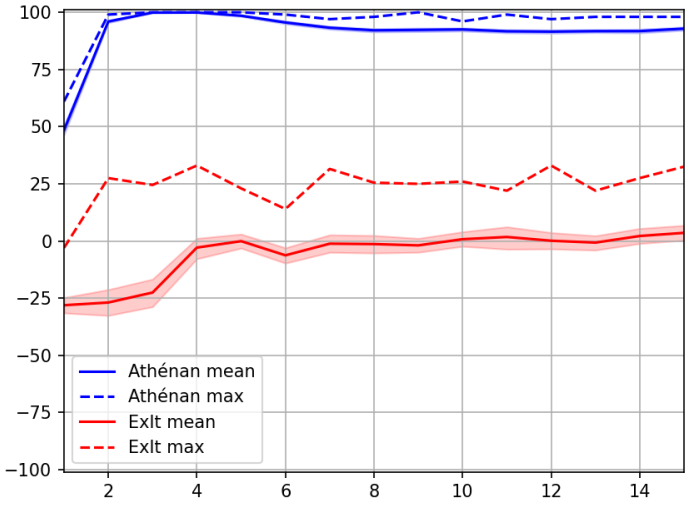

Figure 14: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Othello 10×10 (maximum performance, and mean performance with its bootstrap 95% confidence interval).

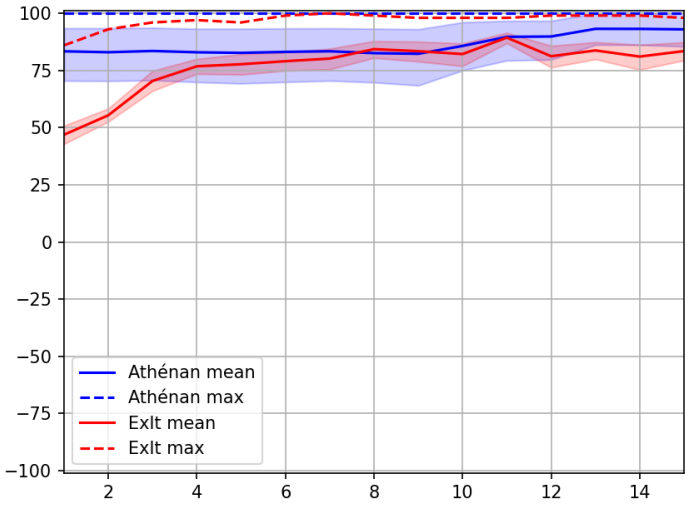

Figure 15: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Outer-Open-Gomoku (maximum performance, and mean performance with its bootstrap 95% confidence interval).

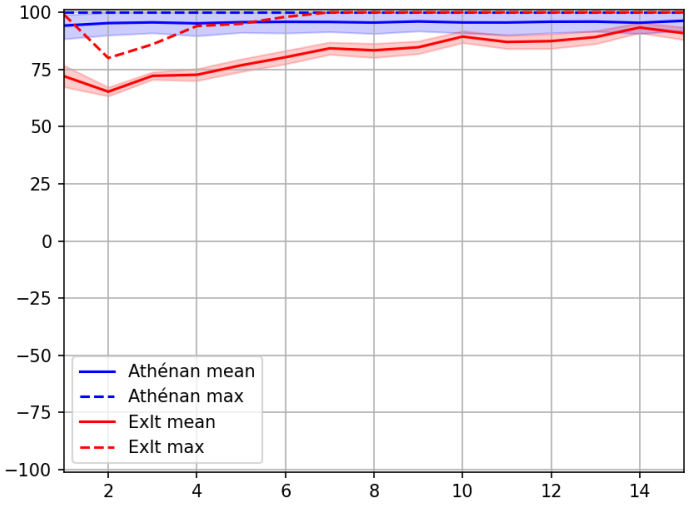

Figure 16: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Connect6 (maximum performance, and mean performance with its bootstrap 95% confidence interval).

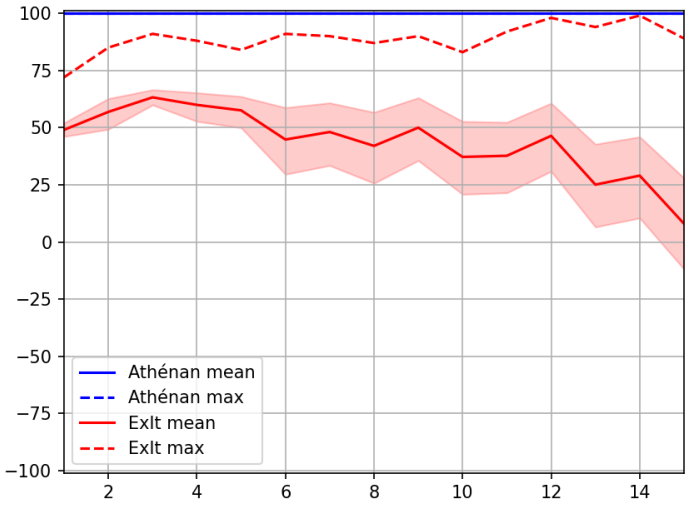

Figure 17: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Havannah 8 (maximum performance, and mean performance with its bootstrap 95% confidence interval).

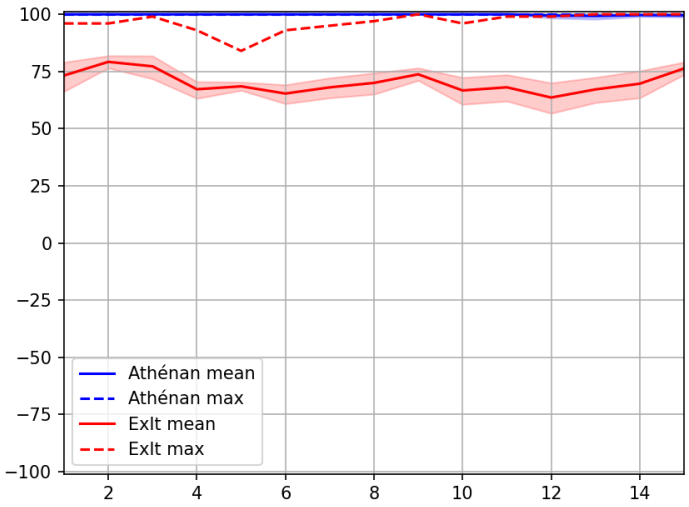

Figure 18: Evolutions over 15 training days of performances of the 40 learning repetitions of Athénan and ExIt against base MCTS for Havannah 10 (maximum performance, and mean performance with its bootstrap 95% confidence interval).

References

Shubhra Aich and Ian Stavness. Global sum pooling: A generalization trick for object counting with small datasets of large images. *arXiv preprint arXiv:1805.11123*, 2018.

Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

Hendrik Baier and Mark HM Winands. Mcts-minimax hybrids with state evaluations. *Journal of Artificial Intelligence Research*, 62:193–231, 2018.

Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.

Elwyn R Berlekamp, John H Conway, and Richard K Guy. *Winning Ways for Your Mathematical Plays, Volume 3*. AK Peters/CRC Press, 2003.

Hans Berliner. The b* tree search algorithm: A best-first proof procedure. In *Readings in Artificial Intelligence*, pages 79–87. Elsevier, 1981.

Yngvi Björnsson and T Anthony Marsland. Learning extension parameters in game-tree search. *Information Sciences*, 154(3-4):95–118, 2003.

Édouard Bonnet, Florian Jamain, and Abdallah Saffidine. On the complexity of connection games. *Theoretical Computer Science*, 644:2–28, 2016.

Cameron Browne. *Hex Strategy: Making the right connections*. AK Peters Natick, MA, 2000.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

Michael Buro. Experiments with multi-probcut and a new high-quality evaluation function for othello. *Games in AI Research*, 34(4):77–96, 1997.

Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial Intelligence*, 134 (1-2):57–83, 2002.

Tristan Cazenave. Residual networks for computer go. *Transactions on Games*, 10(1):107–110, 2018.

Tristan Cazenave and Thomas Fournier. Monte carlo inverse folding. In *Monte Carlo Search International Workshop*, pages 84–99. Springer, 2020.

Tristan Cazenave and Fabien Teytaud. Beam nested rollout policy adaptation. In *ECAI*, 2012.

Tristan Cazenave and Hamza Touzani. Monte carlo inverse rna folding. In *RNA Design: Methods and Protocols*, pages 205–215. Springer, 2024.

Tristan Cazenave, Abdallah Saffidine, Michael John Schofield, and Michael Thielscher. Nested monte carlo search for two-player games. In *The Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, pages 687–693, 2016.

Tristan Cazenave, Yen-Chi Chen, Guan-Wei Chen, Shi-Yu Chen, Xian-Dong Chiu, Julien Dehos, Maria Elsa, Qucheng Gong, Hengyuan Hu, Vasil Khalidov, et al. Polygames: Improved zero learning. *ICGA Journal*, (Preprint):1–13, 2020.

Christopher Clark and Amos Storkey. Training deep convolutional neural networks to play go. In *International Conference on Machine Learning*, pages 1766–1774, 2015.

Quentin Cohen-Solal. Apprendre à jouer aux jeux à deux joueurs à information parfaite sans connaissance. In *Conférence Nationale en Intelligence Artificielle*, 2019.

Quentin Cohen-Solal. Learning to play two-player perfect-information games without knowledge. *arXiv preprint arXiv:2008.01188*, 2020.

Quentin Cohen-Solal. Completeness of unbounded best-first game algorithms. *arXiv preprint arXiv:2109.09468*, 2021.

Quentin Cohen-Solal. Study and improvement of search algorithms in two-players perfect information games. *arXiv preprint arXiv:2505.09639*, 2025.

Quentin Cohen-Solal. Completeness of unbounded best-first minimax and descent minimax. *arXiv preprint arXiv:2603.24572*, 2026a.

Quentin Cohen-Solal. Minibal: Balanced game-playing without opponent modeling. *arXiv preprint arXiv:2603.23059*, 2026b.

Quentin Cohen-Solal. Study and improvement of search algorithms in multi-player perfect-information games. *arXiv preprint arXiv:2604.17378*, 2026c.

Quentin Cohen-Solal and Tristan Cazenave. Descent wins five gold medals at the computer olympiad. *ICGA Journal*, 43(2):132–134, 2021.

Quentin Cohen-Solal and Tristan Cazenave. Athenan wins sixteen gold medals at the computer olympiad. *ICGA Journal*, 45(3), 2023a.

Quentin Cohen-Solal and Tristan Cazenave. Learning to play stochastic two-player perfect-information games without knowledge. *arXiv preprint arXiv:2302.04318*, 2023b.

Quentin Cohen-Solal and Tristan Cazenave. Minimax strikes back. *AAMAS*, 2023c.

Quentin Cohen-Solal and Tristan Cazenave. Athénan wins 11 gold medals at the 2024 computer olympiad. *ICGA Journal*, page 13896911251315102, 2025a.

Quentin Cohen-Solal and Tristan Cazenave. On some improvements to unbounded minimax. *arXiv preprint arXiv:2505.04525*, 2025b.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, pages 72–83, 2007.

Boris Doux, Benjamin Negrevergne, and Tristan Cazenave. Deep reinforcement learning for morpion solitaire. In *Advances in Computer Games*, pages 14–26. Springer, 2021.

Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

Fédération Française d'Othello. Télécharger des programmes d'othello. URL `https://www.ffothello.org/informatique/telecharger-des-programmes/`. [Online; accessed 26-May-2022].

William Fink. An enhancement to the iterative, alpha-beta, minimax search procedure. *ICGA Journal*, 5(1):34–35, 1982.

Arimaa Forum. Arimaa website, 2024. URL `https://arimaa.com/arimaa`.

Chao Gao, Ryan B Hayward, and Martin Müller. Move prediction using deep convolutional neural networks in hex. *Transactions on Games*, 2017.

Chao Gao, Martin Müller, and Ryan Hayward. Three-head neural network architecture for monte carlo tree search. In *International Joint Conference on Artificial Intelligence*, pages 3762–3768, 2018.

Chao Gao, Kei Takada, and Ryan Hayward. Hex 2018: Mohex3hnn over deepezo. *J. Int. Comput. Games Assoc.*, 41(1):39–42, 2019.

Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62, 2005.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *The Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

Richard D Greenblatt, Donald E Eastlake, and Stephen D Crocker. The greenblatt chess program. In *Computer Chess Compendium*, pages 56–66. Springer, 1988.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018.

Shih-Chieh Huang, Broderick Arneson, Ryan B Hayward, Martin Müller, and Jakub Pawlewicz. Mohex 2.0: a pattern-based mcts hex player. In *International Conference on Computers and Games*, pages 60–71. Springer, 2013.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

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

Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

Richard E Korf and David Maxwell Chickering. Best-first minimax search. *Artificial Intelligence*, 84(1-2):299–337, 1996.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436, 2015.

Paweł Liskowski, Wojciech Jaśkowski, and Krzysztof Krawiec. Learning to play othello with deep neural networks. *IEEE Transactions on Games*, 10(4):354–364, 2018.

Michael Lederman Littman. *Algorithms for sequential decision making*. PhD thesis, 1996.

Jacek Mandziuk. *Knowledge-free and learning-based methods in intelligent game playing*, volume 276. Springer, 2010.

Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

Joseph Mellor. *Decision Making Using Thompson Sampling*. PhD thesis, 2014.

Henryk Michalewski, Andrzej Nagórko, and Jakub Pawlewicz. An upper bound of 84 for morpion solitaire 5d. In *CCCG*, pages 270–278, 2016.

Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2009.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

Jan Mycielski. Games with perfect information. *Handbook of Game Theory with Economic Applications*, 1:41–70, 1992.

Andrzej Nagórko. Parallel nested rollout policy adaptation. In *2019 IEEE Conference on Games (CoG)*, pages 1–7. IEEE, 2019.

Andrew Y Ng. Feature selection, l 1 vs. l 2 regularization, and rotational invariance. In *The twenty-first international conference on Machine learning*, page 78. ACM, 2004.

Antonio Norelli and Alessandro Panconesi. Olivaw: Mastering othello without human knowledge, nor a penny. *IEEE Transactions on Games*, 2022.

Jakub Pawlewicz and Ryan B Hayward. Scalable parallel dfpn search. In *International Conference on Computers and Games*, pages 138–150. Springer, 2013.

É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne. Ludii – the ludemic general game system. In G. De Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. BugarÃn, and J. Lang, editors, *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 411–418. IOS Press, 2020.

Stefan Reisch. Hex ist pspace-vollständig. *Acta Informatica*, 15(2):167–191, 1981.

Christopher D Rosin. Nested rollout policy adaptation for monte carlo tree search. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

Jonathan Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990.

Nicol N Schraudolph, Peter Dayan, and Terrence J Sejnowski. Learning to evaluate go positions via temporal difference methods. In *Computational Intelligence in Games*, pages 77–98. Springer, 2001.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017a.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017b.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

David J Slate and Lawrence R Atkin. Chess 4.5 - the northwestern university chess program. In *Chess skill in Man and Machine*, pages 82–118. Springer, 1983.

Dennis JNJ Soemers, Vegard Mella, Cameron Browne, and Olivier Teytaud. Deep learning for general game playing with ludii and polygames. *ICGA Journal*, 43(3):146–161, 2022.

Kei Takada, Hiroyuki Iizuka, and Masahito Yamamoto. Reinforcement learning for creating evaluation function using convolutional neural network in hex. In *2017 Conference on Technologies and Applications of Artificial Intelligence*, pages 196–201. IEEE, 2017.

Kei Takada, Hiroyuki Iizuka, and Masahito Yamamoto. Reinforcement learning to create value and policy functions using minimax tree search in hex. *IEEE Transactions on Games*, 2019.

Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38 (3):58–68, 1995.

Yuandong Tian and Yan Zhu. Better computer go player with neural network and long-term prediction. *arXiv preprint arXiv:1511.06410*, 2015.

H Jaap Van Den Herik and Mark HM Winands. Proof-number search and its variants. In *Oppositional Concepts in Computational Intelligence*, pages 91–118. Springer, 2008.

H Jaap Van Den Herik, Jos WHM Uiterwijk, and Jack Van Rijswijck. Games solved: Now and in the future. *Artificial Intelligence*, 134(1-2):277–311, 2002.

Joel Veness, David Silver, Alan Blair, and William Uther. Bootstrapping from game tree search. In *Advances in Neural Information Processing Systems*, pages 1937–1945, 2009.

Hui Wang, Mike Preuss, Michael Emmerich, and Aske Plaat. Tackling morpion solitaire with alphazero-like ranked reward reinforcement learning. In *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 149–152. IEEE, 2020.

Wikipedia contributors. Computer othello, 2022. URL `https://en.wikipedia.org/wiki/Computer_Othello`. [Online; accessed 26-May-2022].

Mark HM Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-carlo tree search solver. In *International Conference on Computers and Games*, pages 25–36. Springer, 2008.

David J Wu. Designing a winning arimaa program. *ICGA Journal*, 38(1):19–40, 2015.

Georgios N Yannakakis and Julian Togelius. *Artificial intelligence and games*, volume 2. Springer, 2018.

Kenny Young, Gautham Vasan, and Ryan Hayward. Neurohex: A deep q-learning hex agent. In *Computer Games*, pages 3–18. Springer, 2016.