MegaZero: Mastering multi-action games with AlphaZero

Hadrien Crassous, Clément Rodes
Centrale-Supélec, Paris-Saclay University
{hadrien.crassous, clement.rodes}@student-cs.fr

***Abstract*—In multi-action adversarial games, each player's turn consists of several atomic actions. Many strategy games, board games, and video games fall into this category. There are thousands of action sequences available, resulting in a very high branching factor. In this context, finding the best turn is challenging and computationally expensive.**

Reinforcement learning algorithms are state-of-the-art for playing most single-action games without prior human knowledge and for some real-world planning problems, but there are few adaptations for learning multi-action strategic games.

In this paper, we exploit previous work on AlphaZero and on search strategies in multi-action games to teach agents to play multi-action games without prior knowledge.

***Index Terms*—Reinforcement Learning, Adversarial Games, Monte-Carlo Tree Search**

I. Introduction

The challenge of learning adversarial games without prior knowledge is two-fold:

- Learning what game states are promising, i.e. learning a value function.
- Designing a search strategy to select the most optimal action by planning ahead

A. Single-action games

In single action games, computer programs select the next action by using a tree search algorithm, a procedure used to explore and evaluate potential moves. These search strategies construct a tree representing possible future game states by simulating moves for both players. Then they use a value function to evaluate nodes and find the most promising branches. Minimax search with Alpha-Beta pruning [1] has been the first successful technique for strategy games like Chess [2] and Othello [3], with handcrafted evaluation functions. Monte Carlo Tree Search (MCTS) [4] is a popular search algorithm that has been applied to a variety of games with branching factors of up to a few hundred.

The performance of these search strategies have benefited from the recent idea of learning without prior knowledge. In reinforcement learning, the evaluation function is not hard-coded but is learned by a neural network that collects experience by playing games. These frameworks have been the first to achieve superhuman levels of performance in Go, with AlphaGo [5] [6], and in imperfect information games *Stratego* and *DouDizhu*, with DeepNash [7] and DouZero [8].

AlphaZero was itself generalized by MuZero [9]. While AlphaZero requires a black-box model of the game, MuZero learns an abstract model of the game. Essentially, MuZero learns the rules from interactions, regardless of whether the game is single player, multi-player, single-action or multi-action. This allows MuZero to excel also at Atari benchmark and various non-game tasks.

AlphaZero-like frameworks struggle in large action spaces. Scaling MCTS-based RL to complex action spaces is an active research subject, with recent methods being proposed: Warm Start AlphaZero [10], Sequential Monte Carlo Planning [11], EfficientZero [12] for improved sample efficiency, and A0C [13] for continuous action spaces.

B. The challenges of multiple-action games

In turn-based multi-action adversarial games, each turn consists of a sequence of atomic actions instead of just a single action, with much higher branching factors. Board games such as *Arimaa* and *Risk* fall into this category, just like video games such as *Battle of Polytopia*, and *Civilization*. Sometimes, single-action games benefit from being transformed into multi-action games (via SemiSplit [14]) when an action can be decomposed into multiple segments. Unfortunately multi-action games face combinatorial explosion. If each turn, for example, consists of moving nine units with ten available actions each, the resulting branching factor is 10^9.

Naive approaches search a tree in which each edge represents an atomic action instead of a complete turn, but often these trees can not search deeply enough and optimize earlier actions too much compared to later actions, ignoring the opponent's response. Evolutionary approaches like EMCTS [15], OEP [16] and NTBOE [17] [18] have beaten these naive approaches in multi-action games but only with handcrafted heuristics, without proposing a learning framework.

In this paper, we integrate these approaches to the AlphaZero framework, and test their performance in learning multi-action games. As far as we know, this is the first paper that proposes an adaptation of the AlphaZero framework with search strategies that are dedicated to multi-action games.

The main contributions of this work are as follows:

- We propose two testbeds games for light-weight but challenging multi-action adversarial games: Connect4d and Strands
- We propose an adaptation of the AlphaZero learning framework for different search strategies
- We assess the performance of these search strategies on the testbeds games

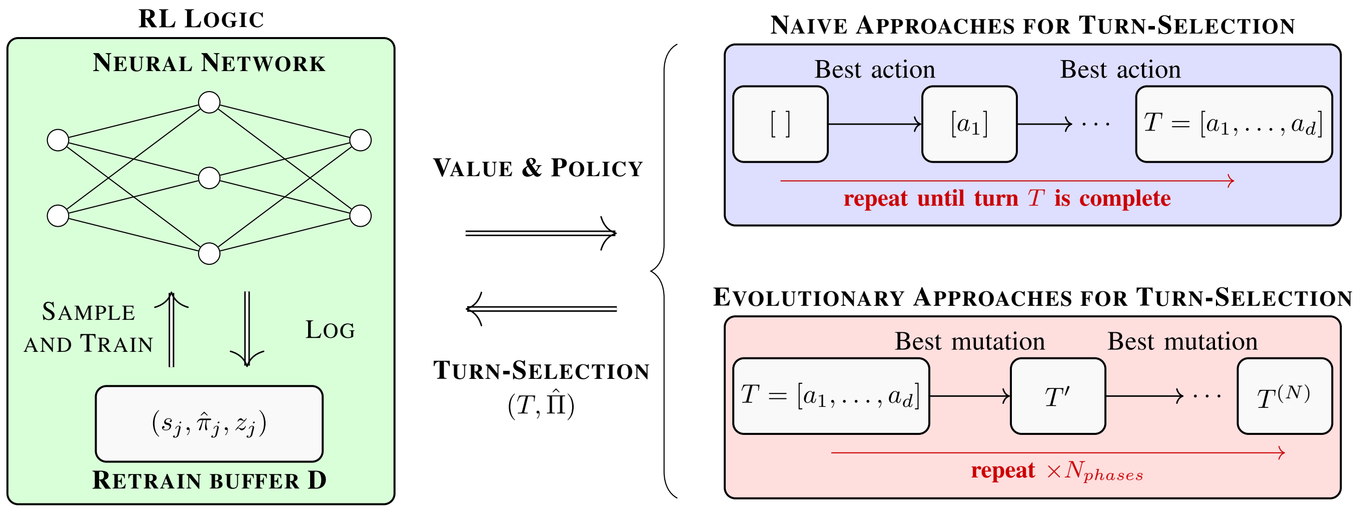

Fig. 1: Different approaches for Turn-Selection in the MegaZero Framework. A turn is either constructed via iterated action selections (naive approaches) or by iterated mutations (evolutionnary approaches)

II. Background and Related Work

We first provide the mathematical formulation that is commonly used in reinforcement learning for adversarial games. Then we detail the state-of-the-art algorithms for reinforcement learning in single-action games and for game search in multi-action games.

A. Mathematical Formulation of Adversarial games and Reinforcement Learning

Our work focuses on the problem of finding the best move in a two-players adversarial and deterministic game, with complete information. The decision-making problem is modeled as a deterministic and finite Markov decision process $M = (\mathcal{S}, \mathcal{S}_p, \mathcal{S}_o, \mathcal{A}, \mathcal{P}, \mathcal{Z}, s_0)$:

- $\mathcal{S} = \mathcal{S}_p \cup \mathcal{S}_o$ denotes the state space
- $\mathcal{S}_p$ denotes the states where the player has to act
- $\mathcal{S}_o$ denotes the states where its opponent has to act
- $\mathcal{A}$ denotes the action space
- $\mathcal{P}$ denotes the transition function mapping $\mathcal{S}_p \times \mathcal{A}$ to $\mathcal{S}_o$ and $\mathcal{S}_o \times \mathcal{A}$ to $\mathcal{S}_p$
- $\mathcal{Z} : \mathcal{S} \to \{-1, 0, 1\}$ denotes the reward function, which maps a state to the outcome z of the game: 0 if the state is not terminal, 1 if the game is won, or -1 if lost
- $s_0 \in \mathcal{S}$ is the starting state of the game.

If the game is multi-action, each player turn consists of several atomic actions. A valid turn $T \in \mathcal{A}$ is made of d atomic actions: $T = (a_{\tau=1}, a_{\tau=2}, \ldots, a_{\tau=d})$, (one atomic action per step τ). The set of atomic actions is denoted $\mathbb{A}$ and the set of complete turns is denoted $\mathcal{A} = \mathbb{A}^d$. The length of the turn is d, also called the dimension of the game.

The objective of RL is to determine a policy $\pi_{player} : \mathcal{S}_p \mapsto \mathcal{A}$ that maximizes the expected game outcome $J(\pi) = \mathbb{E}_{\pi, \pi_{opp}}[z]$, without knowing the opponent policy, $\pi_{opp} : \mathcal{S}_o \mapsto \mathcal{A}$. The player's choices are generally guided by an evaluation function $v : \mathcal{S} \mapsto \mathbb{R}$, which indicates which states are favorable to the player or to the opponent. A search strategy looks ahead in the game to maximize the evaluation of the next states.

Non-RL approaches use handcrafted heuristics $v_h : \mathcal{S} \mapsto \mathbb{R}$ giving approximate values to states. By combining these values with tree search, they select their actions. For elaborate strategic games, good heuristics are difficult to hard-code (like in Go) or they tend to be CPU-intensive tasks. They rely on a compromise between compute efficiency and accurate estimation.

RL approaches train a value function $v_\theta : \mathcal{S} \mapsto \mathbb{R}$ by self-play. It learns what moves or positions are typically good by comparing positions to their game outcome, and they do not need prior human knowledge. They also use tree searches to select actions.

Neural networks have a great computational speed for inference and great approximation capabilities, which solves both problems of hard-coded heuristics. Reinforcement learning allows the collection of an arbitrary amount of data, but training with self-play is generally less stable and more biased than supervised learning.

B. AlphaZero: learning a game without prior knowledge

AlphaZero [19] is a general zero-knowledge learning algorithm for playing two-player adversarial games, without using any human expertise. It achieves superhuman performance in various single-action board games, such as Chess, shogi, and Go.

It is trained entirely through unsupervised self-play, replacing the handcrafted features and heuristic priors commonly used in traditional intelligent programs. Alphazero uses an MCTS tree search, with each node corresponding to an action, to model the game, and it uses upper-confidence trees [4]. The evaluation of each node is made with the value network and the policy network helps guide the exploration. The networks are 2 functions with shared learnable parameters θ: a value network $v_\theta : \mathcal{S} \mapsto \mathbb{R}$ and a policy network $\pi_\theta : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$. At the end of each MCTS search, the number of visits of the root's children helps retrain the policy $\pi_\theta(\text{root})$ (see formulas in section III). Each state visited is stored with the information of the game outcome z and helps improving the value network.

The training process requires to collect a massive amount of very diverse gameplay samples. For instance it took a hundred GPU days to reach a superhuman level of performance in the game of Go. Another bottleneck is that the tree search process does not scale well for large branching rate.

C. *Playing a multi-action game with prior knowledge*

In this subsection, we present prior work on playing multi-action games using handcrafted heuristics.

The most natural approach to perform turn-selection is to construct the turn iteratively by appending the best action proposed by an MCTS search, where the transitions between the nodes represent actions. Other non naive approaches have been proposed, based on evolutionnary approaches (ie treating atomic actions as genes and complete turns as genomes).

Online Evolutionary planning (OEP) [16] is a tree-less evolutionary algorithm that operates through mutation and selection within a population of individuals, each representing a single turn. N-Tuple Bandit Online Evolution (NTBOE) [17] is an improved algorithm for playing multi-action games, inspired by N-Tuple Bandit Evolutionary algorithm (NTBEA) and OEP.

Evolutionary MCTS (EMCTS) is another successful hybrid approach, combining MCTS and evolutionary algorithms. It searches a tree with nodes representing action sequences as genomes, and edges representing mutations of those genomes.

III. Methods

A. *Modification of the AlphaZero algorithm*

We believe that evolutionary approaches can improve AlphaZero by themselves, and we wanted to make the least modifications to the AlphaZero learning scheme, because it has many strengths for learning to play deterministic adversarial games. Thus, we did not change anything but the action-selection method, taking inspiration from the work on tree-searches for multi-action games.

In standard AlphaZero, the MCTS search selects an action a and returns the policy $\hat{\pi}$ improved by the search, based on what are the most visited actions by the MCTS. For MegaZero, we have designed similar processes, that output:

- a selected turn $T = [a_{\tau=1}, \dots, a_{\tau=d}]$
- an improved policy history $\hat{\Pi} = [\hat{\pi}_{\tau=1}, \dots, \hat{\pi}_{\tau=d}]$

We will refer to these processes as "search strategies" in the rest of the paper, and as **TURN-SELECTION** in the pseudo-code 1, inspired by the AlphaZero paper [19].

B. *Vanilla MCTS*

The vanilla MCTS is the most simple search strategy available and it is very similar to the MCTS search strategy in AlphaZero. Vanilla MCTS searches a game tree in which each edge represents an additional action for the turn. The selection policy of this MCTS variant is UCB (upper-confidence bound

Algorithm 1 MegaZero Algorithm

```
Initialize $\theta$ and create retrain buffer $D$
$\pi_\theta, v_\theta$ are the policy and value networks
for iteration $= 1, \dots, I$ do // I iterations
    for episode $= 1, \dots, E$ do // play E games
        for $t = 1, \dots, t_{max}$ do
            $T_t, \hat{\Pi}_t \leftarrow$ TURN-SELECTION$(s_t, \pi_\theta, v_\theta)$
            Execute the atomic actions and update $s_t$
        end for
        Store every states visited as $(s_j, \hat{\pi}_j, z_j)$ in $D$
        (with $z_j$ the game outcome)
    end for
    Sample examples $(s_j, \hat{\pi}_j, z_j)$ from buffer $D$
    Update $\theta$ to jointly minimize value loss $(v_\theta(s_j) - z_j)^2$
and policy loss $CE(\pi_\theta(s_j), \hat{\pi}_j)$ (cross-entropy)
end for
```

[20]), the evaluation of a node is computed with no rollouts via v_θ, and the prior probability of visiting the different children is computed via π_θ. The number of branch evaluations is fixed, and when this exploration budget is consumed, we repeat the phase "select an action and append it to T", until the turn is complete. At phase p, we use the normalized number of visits of the children nodes to compute the improved policy:

$$\hat{\pi}_p = \frac{1}{\Sigma_a N(p,a)} \begin{bmatrix} N(p, a=1) \\ \vdots \\ N(p, a=|\mathbb{A}|) \end{bmatrix} \tag{1}$$

with $N(\tau, a)$ the number of visits of children a from the node of phase p, $T_p = [a_{\tau=1}, ..., a_{\tau=p-1}]$.

C. *Brigde-Burning MCTS*

Bridge-Burning MCTS is a variant that adds pruning to the vanilla MCTS. Instead of using the entire budget to explore from the root $T = [\,]$, the search alternates between exploration and action-selection. In the first phase, a fraction of the budget is used for selecting the first action, the root is updated to $T = [a]$ and the next phase starts with the rest of the budget.

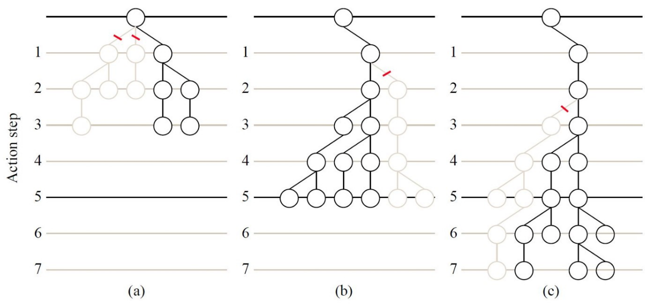

Fig. 2: The "bridge-burning" search strategy (image from [15]) (a) After phase 1, all branches but the best one are pruned, and the root is updated. (b, c) Same thing for phases 2, 3. The process ends after phase d.

As opposed to Vanilla MCTS, which consumes its entire budget on each node—quickly reducing it and leading to increased uncertainty and bias—BB-MCTS distributes the budget evenly across the selection phases. This approach helps maintain confidence in the deeper nodes, and should provide a more accurate estimation. For computing the policy history, the same formula derived from $Equation$ 1 applies.

This search strategy also falls into the category "Naive Approaches " from Figure 1 because it does not involve any evolutionary process.

D. Evolutionary MCTS

Evolutionary MCTS (E-MCTS [15]) searches a game tree in which each node represents a complete sequence $\mathbf{L}$ of actions and each edge represents a mutation, i.e. flipping an action to a different one. The sequence $\mathbf{L}$ can either represent a complete turn, if it has a horizon (size) $H = d$ or a longer sequence composed of multiple turns if $H > d$. In the latter case, the search is called Flexible Horizon Evolutionnary MCTS [21] and it allows to search for the best opponents response.

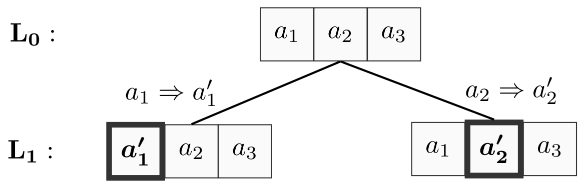

Fig. 3: EMCTS search strategy. Starting from an initial root sequence $\mathbf{L_0}$, the EMCTS algorithm transforms it by mutations ($a_1 \Rightarrow a_1'$) to find a new sequence of actions ($\mathbf{L_1}$).

In contrast with standard MCTS, the number of selection phases that can be performed is not bounded, so we pick a fixed number of phases to perform, as proposed in [15]. We also implemented the option to take a special mutation that preserves the sequence. Another difference is that the branching rate of E-MCTS is about H times greater than the branching rate of MCTS because the number of available mutations is $H \times |\mathbb{A} - 1|$.

We had to take several design choices to adapt this search strategy to the MegaZero framework, in order to define how we use the policy and value networks to guide the search, and how the policy history is computed.

The root sequence $\mathbf{L_0}$ is initialized by following the π_θ distribution, with an additional temperature scaling, to make the initialization greedier. For a matter of compute efficiency, the prior probability of a mutation is computed via π_θ, only once, when generating the root sequence. The same prior is used to repair illegal sequences, when a mutation requires to switch some of the following actions to make the sequence legal. The selection policy and the backpropagation are unchanged. The evaluation of a node is computed by playing the sequence $\mathbf{L}$ and evaluating the resulting state with v_θ, with a minus sign if the mutation represents one of the opponent's action. To compute the improved policy history $\hat{\Pi}$, we use the normalized number of visits of the mutations:

$$\hat{\pi}_\tau = \frac{1}{\Sigma_a N(m = (\tau, a))} \begin{bmatrix} N(m = (\tau, 1)) \\ \vdots \\ N(m = (\tau, |\mathbb{A}|)) \end{bmatrix} \quad (2)$$

where $N(m)$ stands for the number of visits of m as a transition in the tree search.

E. EMCTS with bridge-burning

The same idea of bridge-burning can be used to add pruning to the Evolutionary MCTS, by distributing the exploration budget between the different selection phases.

IV. Testbeds games

A. The requirements for a good testbed

Choosing a suitable testbed game is crucial for evaluating the performance and relevance of the methods. To ensure that the testbed is both challenging and appropriate for reinforcement learning, the following requirements have been considered:

- **High planning demand:** The game should require players to anticipate future outcomes and plan several moves ahead and reward lookahead
- **Absence of reliable heuristics:** As we are interested in learning without prior knowledge, we prioritized games where no heuristic are designed.
- **Simplicity for high throughput:** The game should be fast to simulate, allowing high throughput.

TestBed	Multi-Action	Planning demand	Absence of heuristics	High throughput
Go	×	✓	✓	✓
HeroAcademy [16]	✓	✓	×	×
ASMACAG [17]	✓	✓	×	✓
Connect4d	✓	✓	✓	✓
Strands	✓	✓	✓	✓

TABLE I: Comparison of five testbed game candidates.

We rejected the first three testbed candidates found in the literature, and we worked only on these last 2. In the next subsections, we detail their rules and their specificities.

B. Connect4d

We propose a simple extension of the game Connect4, to make it multi-action, by stacking d boards from Connect4. To play a turn, the player plays sequentially on each one of the d boards. The game ends when 4 tokens are aligned (in 3D space). The game ensures that no immediate win is possible by removing critical directions. A valid 4-token alignment has non-constant depth and non-constant height.

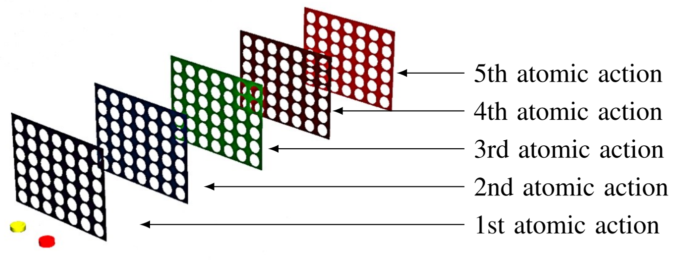

Fig. 4: The Connect4d starting position. In each turn, a player places one token on each of the d boards.

We used $d = 5$ boards of dimension 7×6.

We compare our learning-based solutions to a baseline non-learning player that plays a winning atomic action if it has found one, otherwise a random action.

C. Strands

Strands is a hard-to-master 2-player combinatorial game played on a hexagonal board, with the following rules:

1) To start, Black covers any space marked "2".
2) From then on, starting with White, the players take turns. On your turn, cover up to X empty tiles marked "X". For example, you could cover 3 empty tiles marked "3".
3) The game ends when the board is full. The player with largest connected group of stones wins. If tied, compare the players' second-largest groups, and so on, until you come to a pair which aren't the same size. Whoever owns the larger wins.

In this strategic game, there are thousands of complete turns available, especially in the early game, and playing relevant moves requires to plan ahead.

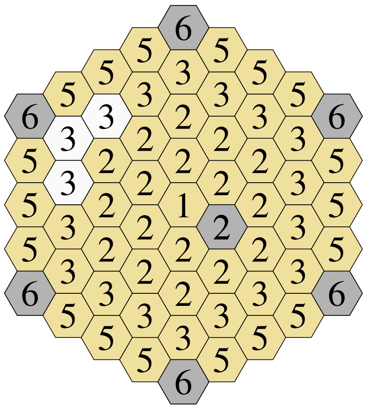

Fig. 5: Example position of Strands after 3 turns. Black started by placing 1 tile on ”2”, white played 3 tiles on ”3”, black played 6 tiles on ”6”.

We compare our learning-based solutions to a ”baseline” non-learning player that looks for the atomic action that maximizes the current advantage, computed via the difference of the size of connected areas. The turns lengths are varying, so we use padding to handle the missing atomic actions. The game can be played on boards with 4 to 6 rings. We worked on a smaller 4-ring board to run more many games.

D. Comparing these two games

The two testbeds have medium-sized action space and dimensions, and they are both challenging with a very high number of legal turns.

	Number of legal actions	d	**Number of legal turns**	**Turns per Game**
Connect4d	7	5	$7^5 = 16807$	$\approx 12^{**}$
5-ring Strands	$\approx 45^*$	6	$\approx 25000^*$	34
4-ring Strands	$\approx 30^*$	4	$\approx 2000^*$	17

TABLE II: Statistics about the testbeds complexity. Approximate values have been obtained by playing random actions the first 5 turns (*) or until the end (**).

The testbeds have significant differences that allowed us to check that our algorithms are valid for different kind of games.

	Fixed-length Games	**Fixed-length Turns**	**Need for sequence repair**
Connect4d	×	✓	✓
Strands	✓	×	×

TABLE III: Differences between the testbeds.

Strands need sequence repair because a legal mutation can lead to an illegal sequence that has to be repaired. A Connect4d can early stop by a winning move, while a Strands Game has a fixed length. A Connect4d turn is always of length d, while turns in Connect4d have different lengths.

V. Experiments

Here, we detail the protocol used to compare the four search strategies: Search Strategies compared Vanilla MCTS (see subsection III-B), Bridge-Burning MCTS (see subsection III-C), Vanilla EMCTS (see subsection III-D) and Bridge-Burning EMCTS (see subsection III-E).

A. Protocol

For the networks, we used a 4-layer ResNet [22] and fully connected output layers. We encoded the game positions by converting them to discrete 2D images with multiple channels.

All four strategies operated under the same exploration budget, meaning that an equal number of network calls across strategies. For each strategy, we conducted one independent learning run, encompassing 50 iterations of 1408 games each, resulting in over 100,000 game samples per iteration.

Every two iterations, the model was evaluated by playing 100 games against a non-learning baseline to generate the learning curves in Figure 6 and Figure 7. After training, we conducted a tournament to evaluate performance, producing the tables in Table IV and Table V. Code, hyperparameters, instructions for reproducing the results and logs are available on our GitHub repository: github.com/Hadrien-Cr/MegaZero. Hyperparameters are shared between the variations, except the UBC exploration factor C which was 8 times smaller in EMCTS to offset the greater branching factor.

B. Key difference with prior experiences

The prior experiences on multi-action search strategies were not in the context of reinforcement learning. The primary difference compared to [15] lies in the time constraints. In reinforcement learning contexts, self-play games must be completed quickly. Our configuration used approximately 0.5 seconds per game, significantly less than the budget used in [15] of 1s per turn. Specifically, we used 250 and 500 MCTS branch evaluations per turn in Connect4d and Strands, respectively, whereas prior work likely employed over ten thousands simulations per turn. In the tournament, we multiplied by 5 the exploration budget because running a tournament is part of an inference phase where the AI has a bigger budget for thinking. In EMCTS, we used a non-flexible horizon during training, and a flexible horizon with $H = 3 \times d$ in the tournament to be more competitive with the naive approaches.

C. Experimental setup

The experiments lasted approximately 30 hours in total, with each run taking 4 hours and each iteration lasting around 5 minutes. All experiments were conducted using a Geforce RTX 3070 Ti GPU and a 6-core Intel i5-12400F processor. The implementation employed multiple AlphaZero workers and batched MCTS for better compute efficiency.

VI. Results

The learning curves Figure 6 and Figure 7 show that all 4 search strategies learn to play these testbeds, beating the non-learning players and improving over time. In the learning curves, we did not notice a significant difference between the learning capabilities of the variants, except the Bridge-Burning MCTS which lags behind. It has 75 % win rate in Connect4d and 85% win rate in Strands, while the other 3 have 90 % win rate in Connect4d and over 95% win rate in Strands.

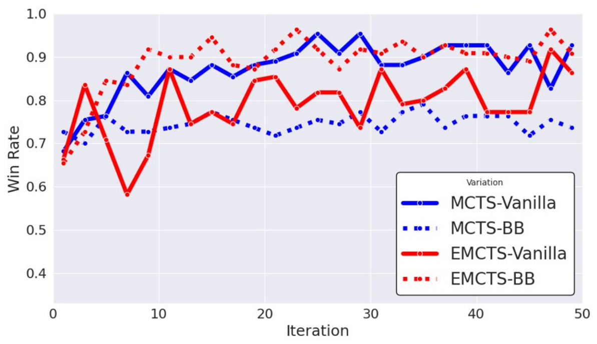

Fig. 6: Learning Curves for all solutions playing Connect4d

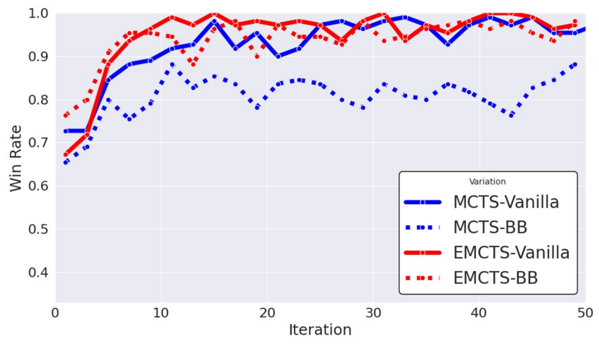

Fig. 7: Learning Curves for all solutions playing Strands

We now present the results of the tournament that compare the variants together. Each match consist of a thousand games with alternating sides. The exploration budget is 5 times bigger than in self-play, and the difference between the variants is even slimmer, all variants approaching game theory optimal plays. Running this tournament took approximately 2 hours. We computed the estimated rating of each variant, with Iterative Luce Spectral Ranking, a metric that is similar to elo. It represents the advantage that a player has on its opponent, a difference in rating of 1 representing a win rate of 75%, while a difference of 3 represents a win rate of 95 %.

Row vs. Col	MCTS-V	MCTS-BB	EMCTS-V	EMCTS-BB	Est. Rating
MCTS-V	\	56.7 %	43.4 %	58.2 %	-0.02
MCTS-BB	43.3%	\	49.0 %	55.4 %	0.07
EMCTS-V	56.6 %	51.0 %	\	50.5 %	**0.09**
EMCTS-BB	41.8 %	44.6 %	49.5 %	\	-0.14

TABLE IV: Winrates of all solutions playing Connect4d.

Row vs. Col	MCTS-V	MCTS-BB	EMCTS-V	EMCTS-BB	Est. Rating
MCTS-V	\	51.9 %	47.2 %	28.8 %	-0.22
MCTS-BB	48.1 %	\	55.3 %	47.1 %	0.0
EMCTS-V	52.8 %	44.7 %	\	50.7 %	-0.02
EMCTS-BB	71.1 %	52.9 %	49.3 %	\	**0.24**

TABLE V: Winrates of all solutions playing Strands.

The main takeaway of these tournaments is that the variants perform very similarly, with very small differences in ratings. The evolutionnary approaches slightly dominate, being twice the best rated, while the Vanilla MCTS has a negative rating in both cases.

VII. Conclusions and Future Work

This paper proposes a new implementation of AlphaZero algorithm for playing turn based adversarial games, where each turn consists of a sequence of multiple actions. Previous reinforcement learning algorithms have demonstrated exceptional performance in single-action games. Some studies have explored the adaptation of these algorithms for multiple-action games, with prior game knowledge. Our algorithm introduces a novel approach for learning multi-action game without prior knowledge.

This approach includes four different search variations based on the Monte Carlo Tree Search : Vanilla MCTS, Bridge Burning MCTS and their evolutionary versions. We proposed two multi-action games called Connect4d and Strands to test them on. Experiments conducted on both games showed the validity of our algorithm, and a comparision between each approaches showed similar learning capabilities.

Future work will address the analysis of the hyperparameters and the impact of a larger exploration budget. Other tasks should be explored, including larger games and non-game domains, in order to find which scenarios discriminate the approaches the best. More work on the interpretability and the policy training is required on EMCTS, because we have taken many naive choices.

In our paper, we did not compare MegaZero to other general adaptations of AlphaZero like MuZero [9] or UNREAL [23]. Comparing our framework to these algorithms and to human performance would help in understanding the strengths and weaknesses.

Our work focused on adapting MCTS and EMCTS to AlphaZero, but an adaptation of OEP and NTBOE could also be explored. Research has shown that they are competitive with MCTS and EMCTS in non-learning contexts, and we guess that these adaptations would lead to similar behaviors but may be easier to tune.

VIII. Acknowledgements

We thank Kevi Aday, Brendan Hansknecht and Surag Nair for their contribution on the AlphaZero general implementation [24]. Our implementation of MegaZero is based on this code.

IX. Division of Work

The work on this article was divided as follows: Hadrien Crassous worked on the original idea, the code, experiment designs and contributed to the writing of all sections. Clement Rodes contributed to the writing of ”Introduction”, ”Background and Related Work”, ”Methods” and ”Conclusion” sections.

References

[1] R. E. Korf and D. M. Chickering, “Best-first minimax search,” *Artificial Intelligence*, vol. 84, no. 1, pp. 299–337, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370295000968

[2] M. Campbell, A. Hoane, and F. hsiung Hsu, “Deep blue,” *Artificial Intelligence*, vol. 134, no. 1, pp. 57–83, 2002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370201001291

[3] S. Thakoor, S. Nair, and M. Jhunjhunwala, “Learning to play othello without human knowledge,” 2016.

[4] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, “A survey of monte carlo tree search methods,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[5] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–489, 01 2016.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, “Mastering the game of go without human knowledge,” *Nature*, vol. 550, pp. 354–359, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:205261034

[7] J. Perolat, B. De Vylder, D. Hennes, E. Tarassov, F. Strub, V. de Boer, P. Muller, J. T. Connor, N. Burch, T. Anthony, S. McAleer, R. Elie, S. H. Cen, Z. Wang, A. Gruslys, A. Malysheva, M. Khan, S. Ozair, F. Timbers, T. Pohlen, T. Eccles, M. Rowland, M. Lanctot, J.-B. Lespiau, B. Piot, S. Omidshafiei, E. Lockhart, L. Sifre, N. Beauguerlange, R. Munos, D. Silver, S. Singh, D. Hassabis, and K. Tuyls, “Mastering the game of stratego with model-free multiagent reinforcement learning,” *Science*, vol. 378, no. 6623, p. 990–996, Dec. 2022. [Online]. Available: http://dx.doi.org/10.1126/science.add4679

[8] D. Zha, J. Xie, W. Ma, S. Zhang, X. Lian, X. Hu, and J. Liu, “Douzero: Mastering doudizhu with self-play deep reinforcement learning,” 2021. [Online]. Available: https://arxiv.org/abs/2106.06135

[9] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, “Mastering atari, go, chess and shogi by planning with a learned model,” *Nature*, vol. 588, no. 7839, p. 604–609, Dec. 2020. [Online]. Available: http://dx.doi.org/10.1038/s41586-020-03051-4

[10] H. Wang, M. Preuss, and A. Plaat, *Warm-Start AlphaZero Self-play Search Enhancements*. Springer International Publishing, 2020, p. 528–542. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-58115-2_37

[11] E. Toledo, M. Macfarlane, D. J. Byrne, S. Singh, P. Duckworth, and A. Laterre, “SMX: Sequential monte carlo planning for expert iteration,” in *ICML 2024 Workshop: Foundations of Reinforcement Learning and Control – Connections and Perspectives*, 2024. [Online]. Available: https://openreview.net/forum?id=a5hWhriatS

[12] W. Ye, S. Liu, T. Kurutach, P. Abbeel, and Y. Gao, “Mastering atari games with limited data,” 2021. [Online]. Available: https://arxiv.org/abs/2111.00210

[13] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker, “A0c: Alpha zero in continuous action space,” 2018. [Online]. Available: https://arxiv.org/abs/1805.09613

[14] J. Kowalski, M. Mika, W. Pawlik, J. Sutowicz, M. Szykuła, and M. H. M. Winands, “Split moves for monte-carlo tree search,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 9, pp. 10 247–10 255, Jun. 2022. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/21265

[15] H. Baier and P. I. Cowling, “Evolutionary mcts for multi-action adversarial games,” *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52989074

[16] N. Justesen, T. Mahlmann, S. Risi, and J. Togelius, “Playing multi-action adversarial games: Online evolutionary planning versus tree search,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. PP, pp. 1–1, 08 2017.

[17] D. Villabrille and R. Montoliu, “NTBOE: a new algorithm for efficiently playing multi-action adversarial games,” in *Actas del I Congreso Español de Videojuegos, Madrid, Spain, December 1-2, 2022*, ser. CEUR Workshop Proceedings, R. Lara-Cabrera and A. J. F. Leiva, Eds., vol. 3305. CEUR-WS.org, 2022. [Online]. Available: https://ceur-ws.org/Vol-3305/paper3.pdf

[18] R. Montoliu, R. D. Gaina, D. Perez-Liebana, D. Delgado, and S. M. Lucas, “Efficient heuristic policy optimisation for a challenging strategic card game.” Springer, Cham, 2020, pp. 403–418.

[19] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, “Mastering atari, go, chess and shogi by planning with a learned model,” *Nature*, vol. 588, no. 7839, p. 604–609, Dec. 2020. [Online]. Available: http://dx.doi.org/10.1038/s41586-020-03051-4

[20] P.-A. Coquelin and R. Munos, “Bandit algorithms for tree search,” 2014. [Online]. Available: https://arxiv.org/abs/1408.2028

[21] H. Baier and P. Cowling, “Evolutionary mcts with flexible search horizon,” *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 14, pp. 2–8, Sep. 2018. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/13023

[22] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[23] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, “Reinforcement learning with unsupervised auxiliary tasks,” 2016. [Online]. Available: https://arxiv.org/abs/1611.05397

[24] S. Thakoor, S. Nair, and M. Jhunjhunwala, “Learning to play othello without human knowledge,” 2016.

APPENDIX
PSEUDO-CODE

Algorithm 2 Search Variant 1: Vanilla MCTS

```
PROCEDURE TURN-SELECTION($s_{root}, \pi_\theta, v_\theta$)
  // Phase 1: Perform MCTS search
  Call MCTS-SEARCH($s_{root}, \pi_\theta, v_\theta$)
  // Phase 2: Pick the turn
  Initialize turn $T$ and policy history $\hat{\Pi}$ to empty lists
  Set node $n$ to the root node with state $s = s_{root}$
  while the turn is not complete:
      $\hat{\pi}(s, a) \leftarrow N(n, a)$ for all atomic actions available $a$ from state $s$ and normalize $\hat{\pi}(n)$
      Use distribution $\hat{\pi}(n)$ to draw an action $a$
      Execute action $a$, Update $n$ and $s$
      Append $a$ to $T$ and $\hat{\pi}(s)$ to $\hat{\Pi}$
  end while
  return $T, \hat{\Pi}$

PROCEDURE MCTS-SEARCH($s_{root}, \pi_\theta, v_\theta$)
  while the budget is not exceeded:
      Set $n$ to the root node with state $s = s_{root}$
      Initialize path to empty list []
      while $n$ is not expanded:
          $a_i \leftarrow UCB\text{-}SELECT\text{-}BEST(n, Q, N, \text{prior=}\pi_\theta)$
          Append $(n, a_i)$ to path
          Execute action $a$, Update $n$ and $s$
      end while
      Evaluate the state $V = v_\theta(s)$
      Backpropagate the value V: for each transition in path, update $Q(n, a)$ and increment $N(n, a)$
      Expand node $n$ by creating children nodes
      Initialize the $Q$ values to 0.
  end while
```

Algorithm 3 Search Variant 2: BB-MCTS

```
PROCEDURE TURN-SELECTION($s_{root}, \pi_\theta, v_\theta$)
  Initialize turn $T$ and policy history $\hat{\Pi}$ to empty lists
  Set node $n$ to the root node with state $s = s_{root}$
  while the turn is not complete:
      // Phase 1: Perform MCTS search
      Call MCTS-SEARCH(node,$\pi_\theta, v_\theta$)
      // Phase 2: Take an atomic action
      $\hat{\pi}(n, a) \leftarrow N(n, a)$ for all atomic actions available $a$
      Normalize $\hat{\pi}(n)$
      Use distribution $\hat{\pi}(n)$ to draw an action $a$
      Execute action $a$, Update $n$ and $s$
      Append $a$ to $T$ and $\hat{\pi}(n)$ to $\hat{\Pi}$
  end while
  return $T, \hat{\Pi}$
```

Algorithm 4 Search Variant 3: FH-EMCTS

```
PROCEDURE TURN-SELECTION($s_{root}, \pi_\theta, v_\theta$)
  // Phase 1: Initialize the sequence $L$
  $L \leftarrow$ INITIALIZE-SEQUENCE($s_{root}, H, \pi_\theta, v_\theta$)
  Initialize $\hat{\Pi}$ to an array of size $(H, |\mathbb{A}|)$ filled with zeros.
  // Phase 2: Perform EMCTS search
  Call EMCTS-SEARCH($s_{root}, \pi_\theta, v_\theta, L, \hat{\Pi}$)
  // Phase 3: Pick the turn
  T is the first turn from the sequence L
  Normalize $\hat{\Pi}$
  return $T, \hat{\Pi}$

PROCEDURE EMCTS-SEARCH($s_{root}, \pi_\theta, v_\theta, L_{root}, \hat{\Pi}$)
  while the budget is not exceeded:
      Set $n$ to the root node with sequence $L = L_{root}$
      Initialize path to empty list []
      while $n$ is not expanded:
          $m \leftarrow UCB\text{-}SELECT\text{-}BEST(n, Q, N, \text{prior=}\pi_\theta)$
          Append $(n, m)$ to path
          Mutate the sequence $L$ and update $n$
      end while
      Sequentially execute the sequence $L$ starting from $s_{root}$
      Evaluate the resulting state $V = v_\theta(s)$
      Backpropagate the value V: for each transition in path, update $Q(n, m)$, increment $N(n, m)$ and increment $\hat{\Pi}(m)$,
      Expand node $n$ by creating children nodes
      Initialize the children $Q$ values to 0.
  end while
```