Improving AlphaZero Using Monte-Carlo Graph Search

Johannes Czech[1], **Patrick Korus**[1], **Kristian Kersting**[1, 2, 3]
[1] Department of Computer Science, TU Darmstadt, Germany
[2] Centre for Cognitive Science, TU Darmstadt, Germany
[3] Hessian Center for Artificial Intelligence (hessian.AI), Darmstadt, Germany
johannes.czech@cs.tu-darmstadt.de, patrick.korus@stud.tu-darmstadt.de, kersting@cs.tu-darmstadt.de

Abstract

The *AlphaZero* algorithm has been successfully applied in a range of discrete domains, most notably board games. It utilizes a neural network that learns a value and policy function to guide the exploration in a Monte-Carlo Tree Search. Although many search improvements such as graph search have been proposed for Monte-Carlo Tree Search in the past, most of them refer to an older variant of the Upper Confidence bounds for Trees algorithm that does not use a policy for planning. We improve the search algorithm for *AlphaZero* by generalizing the search tree to a directed acyclic graph. This enables information flow across different subtrees and greatly reduces memory consumption. Along with Monte-Carlo Graph Search, we propose a number of further extensions, such as the inclusion of ϵ-greedy exploration, a revised terminal solver and the integration of domain knowledge as constraints. In our empirical evaluations, we use the *CrazyAra* engine on chess and crazyhouse as examples to show that these changes bring significant improvements to *AlphaZero*.

Introduction

The planning process of most humans for discrete domains is said to resemble the *AlphaZero* Monte-Carlo Tree Search (MCTS) variant (Silver et al. 2017), which uses a guidance policy as its prior and an evaluation function to distinguish between good and bad positions (Hassabis et al. 2017). The human reasoning process, however, may not be as stoic and structured as a search algorithm running on a computer, and humans are typically not able to process as many chess board positions in a given time. Nevertheless, humans are quite good at making valid connections between different subproblems and to reuse intermediate results for other subproblems. In other words, humans not only look at a problem step by step but are also able to jump between sequences of moves, so called trajectories; they seem to have some kind of *global memory buffer* to store relevant information. This gives a crucial advantage over a traditional tree search, which does not share information between different trajectories, although an identical state may occur.

Triggered by this intuition, the search tree is generalized to a Directed Acyclic Graph (DAG), yielding Monte-Carlo Graph Search (MCGS) (Saffidine, Cazenave, and Méhat 2012). The search in the DAG follows the scheme of the Upper Confidence bounds for Trees (UCT) algorithm (Auer, Cesa-Bianchi, and Fischer 2002), but employs a modified forward and backpropagation procedure to cope with the graph structure. Figure 1 illustrates, how nodes with more than one parent, so called transposition nodes, allow to share information between different subtrees and vastly improve memory efficiency.

More importantly, a graph search reduces the amount of neural network evaluations that normally take place when the search reaches a leaf node. Together, MCGS can result in a substantial performance boost, both for a fixed time con-

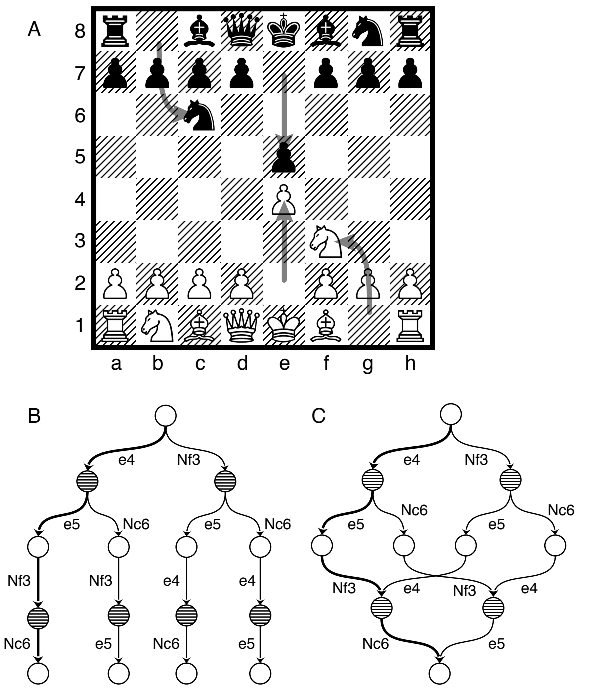

Figure 1: It is possible to obtain the King's Knight Opening (A) with different move sequences (B, C). Trajectories in **bold** are the most common move order to reach the final position. As one can see, graphs are a much more concise representation.

straint and for a fixed amount of evaluations. This is demonstrated in an empirical evaluation on the example of chess and its variant crazyhouse.

Please note, that we intentionally keep our focus here on the planning aspect of *AlphaZero*. We provide a novel planning algorithm that is based on DAGs and comes with a number of additional enhancements. Specifically, our contributions for improving the search of *AlphaZero* are as follows:

1. Transforming the search tree of the *AlphaZero* framework into a DAG and providing a backpropagation algorithm which is stable both for low and high simulation counts.
2. Introducing a terminal solver to make optimal choices in the tree or search graph, in situations where outcomes can be computed exactly.
3. Combining the UCT algorithm with ϵ-greedy search which helps UCT to escape local optima.
4. Using Q-value information for move selection which helps to switch to the second best candidate move faster if there is a gap in their Q-values.
5. Adding constraints to narrow the search to events that are important with respect to the domain. In the case of chess these are trajectories that include checks, captures and threats.

We proceed as follows. We start off by discussing related work. Then, we give a quick recap of the PUCT algorithm (Rosin 2011), the variant of UCT used in *AlphaZero*, and explain our realisation of MCGS and each enhancement individually. Before concluding, we touch upon our empirical results. We explicitly exclude Reinforcement Learning (RL) and use neural network models with pre-trained weights instead. However, we give insight about the stability of the search modifications by performing experiments under different simulation counts and time controls.

Related Work

There exists quite a lot of prior work on improving UCT search. A comprehensive overview of these techniques is covered by Browne et al. (2012). However, these earlier techniques focus on a version of UCT, which only relies on a value without a policy approximator. Consequently, some of these extensions became obsolete in practice, providing an insignificant improvement or even deteriorated the performance. Each of the proposed enhancements also increases complexity, and most require a full re-evaluation when they are used for PUCT combined with a deep neural network.

Saffidine, Cazenave, and Méhat (2012) proposed to use the UCT algorithm with a DAG and suggested an adapted UCT move selection formula (1). It selects the next move a_t with additional hyperparameters $(d_1, d_2, d_3) \in \mathbb{N}^3$ by

$$a_t = \text{argmax}_a \left(\mathrm{Q}_{d_1}(s_t, a) + U_{d_2,d_3}(s_t, a)\right) \,, \qquad (1)$$

where

$$U_{d_2,d_3}(s_t, a) = c_{\text{puct}} \cdot \sqrt{\frac{\log \sum_b N_{d_2}(s_t, b)}{N_{d_3}(s_t, a)}} \,. \qquad (2)$$

The values for d_1, d_2 and d_3 relate to the respective depth and were chosen either to be 0, 1, 2 or ∞. Their algorithm was tested on several environments, including a toy environment called `LEFTRIGHT`, on the board game `HEX` and on a 6×6 `GO` board using 100, 1000 and 10 000 simulations, respectively. Their results were mixed. Depending on the game, a different hyperparameter constellation performed best and sometimes even the default UCT-formula, corresponding to $(d_1 = 0, d_2 = 0, d_3 = 0)$, achieved the highest score.

Choe and Kim (2019) and Bauer, Patten, and Vincze (2019) build upon the approach of Saffidine, Cazenave, and Méhat (2012). More specifically, Choe and Kim (2019) modified the selection formula (2) while reducing the number of hyperparameters to two and applied the algorithm to the card-game Heartstone. Similarly, Bauer, Patten, and Vincze (2019) employed a tuned selection policy formula of (2) which considers the empirical variance of rewards and used it for the task of object pose verification.

Graphs were also studied as an alternative representation in a trial-based heuristic tree search framework by Keller and Helmert (2013) and further elaborated by Keller (2015). Moreover, Childs, Brodeur, and Kocsis (2008) explored different variants for exploiting transpositions in UCT. Finally, Pélissier, Nakamura, and Tabata (2019) made use of transpositions for the task of feature selection. All other directed acyclic graph search approaches have in common that they were developed and evaluated in UCT-like framework which does not include an explicit policy distribution.

Also the other enhancement suggested in the resent paper, namely the terminal solver, extends an already existing concept. Chen, Chen, and Lin (2018) build up on the work by Winands, Björnsson, and Saito (2008) and presented a terminal solver for MCTS which is able to deal with drawing nodes and allows to prune losing nodes.

Finally, challenged by the *Montezuma's Revenge* environment, which is a hard exploration problem with sparse rewards, Ecoffet et al. (2021) described an algorithm called *Go-Explore* which remembers promising states and their respective trajectories. Subsequently, they are able to both return to these states and to robustify and optimize the corresponding trajectory. Their work only indirectly influenced this paper, but gave motivation to abandon the search tree structure and to keep a memory of trajectories in our proposed methods.

The PUCT Algorithm

The essential idea behind the UCT algorithm is to iteratively build a search tree in order to find a good choice within a given time limit (Auer, Cesa-Bianchi, and Fischer 2002). Nodes represent states s_t and edges denote actions a_t for each time step t. Consider e. g. chess. Here, nodes represent board positions and edges denote legal moves that transition to a new board position. Now, each iteration of the tree search consists of three steps. First, selecting a leaf node, then expanding and evaluating that leaf node and, finally, updating the values $V(s_t)$ and visit counts $N(s_t)$ on all nodes in the trajectory, from the selected leaf node up to the root node.

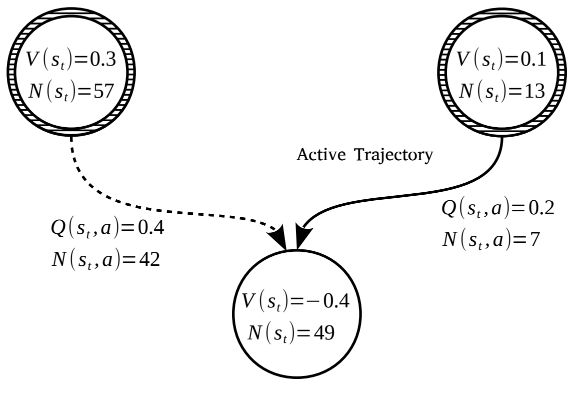

Figure 2: Scenario of accessing a child node on two possible trajectories from different parent states. The proposed data structure stores the visits and Q-values both on the edges and in the nodes.

To narrow down the search, the PUCT algorithm, introduced by Rosin (2011) and later refined by Silver et al. (2017), makes use of a prior policy $\mathbf{p}$. At each state s_t, for every time step t, a new action a_t is selected according to the UCT-formula (3) until either a new unexplored state s^* or a terminal node s_T is reached, i.e.,

$$a_t = \text{argmax}_a \left(\text{Q}(s_t, a) + U(s_t, a)\right) \ , \quad (3)$$

$$\text{where} \quad U(s_t, a) = c_{\text{puct}} P(s_t, a) \frac{\sqrt{\sum_b N(s_t, b)}}{1 + N(s_t, a)} \ . \quad (4)$$

The neural network f_θ then evaluates the new unexplored state s^*. Every legal action a_i is assigned a policy value $P(s, a_i)$ and the state evaluation $\widetilde{v}$ is backpropagated along the visited search path.

If we encounter a terminal state s_T, then the constant evaluation of either -1, $+1$ or 0 is used instead. In the case of a two player zero-sum game, the value evaluation $\widetilde{v}$ is multiplied by -1 after each turn. The respective Q-values (Sutton and Barto 2018) are updated by a Simple Moving Average (SMA):

$$Q'(s_t, a) = Q(s_t, a) + \frac{1}{n}\left[\widetilde{v} - Q(s_t, a)\right] \ . \quad (5)$$

Unvisited nodes are treated as losses and assigned a value of -1. Moreover, the Q- and U-values are weighted according to the parameter c_{puct} which is scaled with respect to the number of visits of a particular node:

$$c_{\text{puct}}(s) = \log \frac{\sum_a N(s, a) + c_{\text{puct-base}} + 1}{c_{\text{puct-base}}} + c_{\text{puct-init}} \ . \quad (6)$$

We choose $c_{\text{puct-base}}$ to be 19652 and a $c_{\text{puct-init}}$ value of 2.5 which is based on the choice of (Silver et al. 2017) but scaled to a larger value range.

Monte-Carlo Graph Search

The main motivation of having a DAG structure is to share computation from different branches and to reduce memory allocation. If we reach a state, which has already been explored from a different subtree, we can make use of already computed value estimates instead of solving the same subproblem from scratch. Specifically, we detect transpositions, i.e., that the same position was reached on different trajectories, by using a hash key function. If this is the case, we create an edge between the current node and the pre-existing subtrees. We incorporate the step counter within the transposition hash key. This way we cannot reach a state with the same key more than once within a trajectory. Therefore, cycles cannot occur.

However, there are other problems to consider. A naïve implementation would straightforwardly share information between different subtrees by creating a deep copy of previous neural network evaluation results. Indeed, this allows reusing previous evaluations and reducing the amount of neural network requests. Hence, computational resources are not wasted on reevaluating already known information, but instead, new information is gained on each network evaluation. Unfortunately, additional memory is required to copy the policy distribution and value of the pre-existing node and the resulting tree structure uses at least as much memory as the vanilla MCTS. Moreover, the backpropagation can be conducted only on the traversed trajectory or on all possible trajectories on which the transposition is accessible.

Updating all trajectories has both a bad scaling behaviour as well as leading to negative side-effects (Saffidine, Cazenave, and Méhat 2012). If we only update the traversed trajectory, we encounter the problem of leaking information instead: each Q-value of the parent nodes of a tranposition node bases its approximation only on a subset of the value information within the subtree, but uses the full subtree for node selection. Therefore, it loses the information which is incorporated in the Q-value of the other parent nodes of the transposition node. This phenomenon occurs more frequently as the number of simulations increases and makes this approach unstable.

In the following, we develop a solution for both of the key issues: we address the memory problem, while preventing information leaks. To this end, MCGS will be explained and evaluated using PUCT for move selection.

Data-Structure

There are multiple possible data structures to realize a DAG and to conduct backpropagation in a DAG. Q-values and visits can be stored on the edges, on the nodes or even both.

We propose to store the Q-values on both to avoid information leaking. Indeed, this is accompanied with a higher memory consumption and a higher computational effort. However, it also allows us to differentiate between the current belief of the Q-value on the edge and the more precise value estimation on the transposition node. The value estimation of the next node is generally more precise or has the same precision than the incoming Q-value on the edge because $N(s_t, a) \leq N(s_{t+1})$.

As our data structure, we keep all trajectories of the current mini-batch in memory, because these will later be used during the backpropagation process.

Algorithm 1: Node Selection and Expansion of MCGS

```
Data: rootNode, $s_0$, $Q_\epsilon$
Result: trajectory, value
node ← rootNode;
$s_t \leftarrow s_0$;
while node is not leaf do
    (nextNode, edge) ← select node using (3);
    append (node, edge) to trajectory;
    if nextNode is transposition then
        $Q_\delta \leftarrow Q(s_t, a) - V^*(s_{t+1})$;
        if $Q_\delta > Q_\epsilon$ then
            $Q_\phi(s_t, a) \leftarrow$
              $N(s_t, a) \cdot Q_\delta(s_t, a) + V^*(s_{t+1})$;
            $Q'_\phi(s_t, a) \leftarrow$
              $\max(V_{\text{min}}, \min(Q_\phi(s_t, a), V_{\text{max}}))$;
            value ← $Q'_\phi$;
            return trajectory, value;
    if nextNode is terminal then
        value ← nextNode.value;
        return trajectory, value;
    node ← nextNode;
    $s_t \leftarrow$ apply action edge.$a$ on $s_t$;
expand node;
(node.$v$, node.$\mathbf{p}$) ← $f_\theta(s_t)$;
value ← node.$v$;
return trajectory, value;
```

Algorithm 2: Backpropagation of MCGS

```
Data: trajectory, value
Result: updated search graph
qTarget ← NaN;
while (node, edge) in reverse(trajectory) do
    if qTarget != NaN then
        $Q_\delta \leftarrow Q(s_t, a) -$ qTarget;
        $Q_\phi(s_t, a) \leftarrow N(s_t, a) \cdot Q_\delta(s_t, a) + V^*(s_{t+1})$;
        $Q'_\phi(s_t, a) \leftarrow$
          $\max(V_{\text{min}}, \min(Q_\phi(s_t, a), V_{\text{max}}))$;
        value ← $Q'_\phi$;
    else
        value ← −value;
    update edge.$q$ with value;
    edge.$n$++;
    update node.$v$ with value;
    node.$n$++;
    if node is transposition then
        qTarget ← −node.$v$;
    else
        qTarget ← NaN
```

Selection, Expansion and Backpropagation

With the data structure at hand, we can now realise the value update as well as modify backpropagation correspondingly. Consider the situation depicted in Figure 2. If the next node of a simulation trajectory is a transposition node, i.e., a node that has more than one parent node, we define our target $V^*(s_{t+1})$ for the Q-value $Q(s_t, a)$ as the inverse of the next value[1] estimation:

$$V^*(s_t) = -V(s_t). \tag{7}$$

Now, we calculate the residual of our current Q-value belief $Q(s_t, a)$ compared to the more precise value estimation $V^*(s_{t+1})$ thereafter. We define this residual as Q_δ:

$$Q_\delta(s_t, a) = Q(s_t, a) - V^*(s_{t+1}), \tag{8}$$

that measures the amount of the current information leak.

If our target value $V^*(s_{t+1})$ has diverged from our current belief $Q(s_t, a)$, e. g. $|Q_\delta| > 0.01$, we already have a sufficient information signal and do not require an additional neural network evaluation. Consequently, we stop following the current trajectory further and avoid expensive neural network evaluations which are unlikely to provide any significant information gain. We call Q_ϵ the hyper-parameter for the value of 0.01 and it remains the only hyper parameter in this algorithm. If, however, $|Q_\delta| \leq Q_\epsilon$, then we iteratively apply the node selection formula (3) of the PUCT algorithm to reach a leaf-node. Otherwise, in case of $|Q_\delta| > Q_\epsilon$, we backpropagate a value that does not make use of a neural network evaluation and brings us close to $V^*(s_{t+1})$. This correction value, denoted as $Q_\phi(s_t, a)$, is

$$Q_\phi(s_t, a) = N(s_t, a) \cdot Q_\delta(s_t, a) + V^*(s_{t+1}) \tag{9}$$

and can become greater than Q_{max} or smaller than Q_{min} for large $N(s_t, a)$. To ensure that we backpropagate a well-defined value, we clip our correction value to be within $[V_{min}, V_{max}]$, i.e.,

$$Q'_\phi(s_t, a) = \max(V_{\text{min}}, \min(Q_\phi(s_t, a), V_{\text{max}})). \tag{10}$$

We also just incorporate the correction procedure of (9) and (10) into the backpropagation process after every transposition node. A compact summary of the forward- and back-propagation procedure is shown in Algorithm 1 and 2.

[1]For a single player game, $V^*(s_{t+1})$ is equivalent to $V(s_{t+1})$.

Discussion

Our MCGS algorithm makes several assumptions. First, we assume the state to be Markovian, because the value estimation of trajectories with an alternative move ordering is shared along transposition nodes. This assumption, however, might be violated if history information in the neural network input representation. In practice, for environments such as chess that are theoretically Markovian, this did not result in a major issue.

As previously mentioned, our data-structure employs redundant memory and computation for nodes that are currently not transposition nodes but may become transposition nodes in the future. However, it should be considered that the bottlekneck of *AlphaZero's* PUCT algorithm is actually the neural network evaluation, typically executed on a Graphics Processing Unit (GPU). This may explain our observation that spending a small overhead on more CPU computation did not result in an apparent speed loss.

Bear in mind that a node allocates memory for several statistics. While value and visits are scalar, a node also has to hold the policy distribution for every legal move even if it will never be used through the search. The memory consumption of the latter is larger by orders of magnitude. As a consequence, we observe a memory reduction of 30 % to 70 % depending on the position and the resulting amount of transposition nodes.

Finally, we want to discuss how our changes to the *AlphaZero* search affect optimality guarantees. If no transpositions occur, our algorithm behaves identically to the original (Silver et al. 2017). This statement holds true for all subtrees near the leaf nodes, which do not hold any transposition nodes, yet. Thus, if for the first occurrence of a transposition node, which leads to one of these trees, the same guarantees hold as in the original, the overall optimality can be established by recursion. There are two cases in the node selection phase that we need to consider. First, the Q-value on the edge to the transposition node is the same as the value of the transposition node. In this case, the evaluation, which is later backpropagated, remains unaffected.

Second, the value of the transposition node has diverged from the Q-value to the transposition node. This can happen if simulations have reached the transposition node from a different trajectory before the current edge has been revisited. In this case, we alter the evaluation during backpropagation to the expectation of all skipped evaluations. Because this expectation represents a more accurate evaluation than a single evaluation, the convergence properties still apply.

Further Enhancements of MCGS

Beyond the MCGS algorithm, we propose a set of additional independent enhancements to the *AlphaZero* planning algorithm. In the following, we describe these methods in detail.

Improved Discovering of Forcing Trajectories

Due to sampling, MCGS may miss a critical move or sample a move again and again even if it has been explored already with an exact "game-theoretical" value. To help pruning losing lines completely and increasing the chance of finding an exact winning move sequence, we now introduce a terminal solver into the planning algorithm (Chen, Chen, and Lin 2018). Doing so allows early stopping, and to select the so far known shortest line when in a winning position. It also provides a stronger learning signal during RL. Accordingly, the longest line can be chosen, when in a losing position, and if a step counter is computed.

Specifically, we add an identifier called END_IN_PLY, which keeps track of the number of steps until a terminal node is reached. When a new node has been solved, END_IN_PLY is assigned the same value as a designated child node and is then incremented by one. In case of a LOSS, the terminal solver chooses the child node with the highest END_IN_PLY value and the smallest value in case of a WIN or DRAW. The variable UNKNOWN_CHILDREN_COUNT describes the number of child nodes where the state is UNKNOWN. Besides that, we add three new node states TB_WIN, TB_LOSS, and TB_DRAW, which are used to determine forced lines to reach a table base position. If a node has been proven to be a TB_LOSS/LOSS, then we prune its access by setting its Q-value to $-\infty$ and policy value to 0.0.

Algorithm 3: Backpropagation of Exact-win-MCTS 2.0

```
if has-loss-child then
    mark WIN;
    parent.UNKNOWN_CHILDREN_COUNT--;
    END_IN_PLY ← $\min_{\text{child}}$(END_IN_PLY$_{\text{child}}$) + 1;
if UNKNOWN_CHILDREN_COUNT == 0 then
    if has-draw-child then
        mark DRAW;
        parent.UNKNOWN_CHILDREN_COUNT--;
        END_IN_PLY ← $\min_{\text{child}}$(END_IN_PLY$_{\text{child}}$)
          + 1;
    else
        mark LOSS;
        parent.UNKNOWN_CHILDREN_COUNT--;
        END_IN_PLY ← $\max_{\text{child}}$(END_IN_PLY$_{\text{child}}$)
          + 1;
```

As argued in Table 1, this terminal solver gives a significant advantage over other terminal solvers. Optimality is preserved because only nodes are pruned that lead to losing terminal nodes using a forced sequence of moves. The pseudo-code of the backpropagation for determining the node states WIN, LOSS and DRAW is shown in Algorithm 3. The computation of solving node states that can reach a table base position by force is carried out analogously to the aforementioned one. As soon as we reach a tablebase position, we can still make use of our value evaluation of our neural network model, in order to converge to a winning terminal node faster. If we only used the tablebase value evaluation instead, we would encounter the problem of reaching a plateau and our search would be unable to differentiate between different winning or losing node states.

In order to accelerate the convergence to terminal nodes, we decouple the forward and backpropagation process of terminal trajectories and allow a larger amount of terminal trajectories during the creation of a mini-batch.

Random Exploration to Avoid Local Optima

The ϵ-greedy search is a well known exploration mechanism which is often applied in RL algorithms such as Q-learning (Watkins and Dayan 1992). The idea behind ϵ-greedy search is to follow the so far best known trajectory and to explore a different action with a probability ϵ_{greedy} instead. Over time, the influence of the prior policy in the PUCT algorithm diminishes and more simulations are spend on the action with the current maximum Q-value. Formula (3) is meant to provide a mechanism to balance exploiting known values and exploring new, possibly unpromising nodes. In fact, it is proven that this formula will find an optimal solution with a sufficient amount of simulations (Auer, Cesa-Bianchi, and Fischer 2002).

However, in practice, we see that a lot of the times the

Table 1: Comparison of different terminal solver implementations.

	Exact-win-MCTS 2.0 (Ours)	**Exact-win-MCTS**	**MCTS-Solver+MCTS-MB**
Node States	`WIN`, `LOSS`, `DRAW`, `UNKNOWN`	`WIN`, `LOSS`, `DRAW`, `UNKNOWN`	`WIN` and `LOSS`
Optional Node States	`TB_WIN`, `TB_LOSS`, `TB_DRAW`	-	-
Member Variables	`UNKNOWN_CHILDREN_COUNT`, `END_IN_PLY`	`UNKNOWN_CHILDREN_COUNT`	-
Nodes have been proven	Prune	Prune	May revisit
Draw games	✓	✓	✗
Supports Tablebases	✓	✗	✗
Selects shortest found mate	✓	✗	✗

fully deterministic PUCT algorithm from *AlphaZero* needs an unpractical amount of simulations to revisit optimal actions where the value evaluations of the first visits are misleading. This motivates adding stochasticity to sooner escape such scenarios. Breaking the rule of full determinism and employing some noise is also motivated by optimization algorithms such as SGD (Srivastava et al. 2014) as well as dropout (Srivastava et al. 2014) that improve convergence and robustness. In the context of RL it was suggested by Silver et al. (2017) to apply Dirichlet noise $\mathrm{Dir}(\alpha)$ on the policy distribution $\mathbf{p}$ of the root node to increase exploration. However, this technique has several disadvantages when being employed in a tournament setting particularly with large simulation budgets.

Utilizing a static Dirichlet noise at the root node generally increases the exploration rate of a few unpromising actions because the amount of suboptimal actions is often larger than relevant actions. Therefore, it is more desirable to apply uniform noise. Additionally, it is favorable to apply such noise not only at root level but at every level of the tree. Such additional uniform noise is what underlies the ϵ-greedy algorithm.

PUCT, UCT and ϵ-greedy have a lot in common by trying to find a good compromise for the exploration-exploitation dilemma. PUCT and UCT usually converge faster to the optimal trajectory than ϵ-greedy (Sutton and Barto 2018). ϵ-greedy selects actions greedily but with static uniform noise, instead. Therefore, it provides a suitable mechanism to overcome local optima where PUCT gets stuck. The ϵ-greedy algorithm can be straightforwardly implemented at the root level of the search as there is no side-effect of sampling a random node from the root node, except of potentially wasting simulations on unpromising actions. However, if the mechanism is utilized at nodes deeper in the tree, we disregard the value expectation formalism and corrupt all its parent nodes on its trajectory.

Following this regime, we propose to use disconnected trajectories for ϵ-greedy exploration. Specifically, a new exploration trajectory is started if a random variable uniformly drawn from $[0, 1]$ is $\leq \epsilon$. Next, we determine the depth on which we want to start the branching. We want to generally, prefer branching at early layers. Therefore, we draw a new random variable r_2 and exponentially reduce the chance of choosing a layer with increasing depth

$$\text{depth} = -\log_2(1 - r_2) - 1 \; . \tag{11}$$

Unexplored nodes are expanded in descending order. Usually, the policy is ordered already, to allow a more efficient dynamic memory allocation and node-selection formula. Therefore this step does not require an additional overhead.

Note that, if we use random exploration at the root node, the guarantees of the original algorithm are not violated as it simply represents a form of root parallelization. Indeed, this statement does not apply to higher depths as it leads to an intentional information leak. However, we aim to bound the amount of information leaked by choosing a low probability for random exploration.

Using Q-Value information for Move Selection

The default move selection policy π is based on the visits distribution of the root node which can optionally be adjusted by a temperature parameter τ,

$$\pi(a|s_0) = N(s_0, a)^{\frac{1}{\tau}} / \Big(\sum\nolimits_b N(s_0, b)^{\frac{1}{\tau}} \Big). \tag{12}$$

Including the Q-values for move selection can be beneficial because the visits and Q-value distribution have a different convergence behaviour. The visit distribution increases linearly over time whereas the Q-value can converge much faster if the previously best move found was found to be losing. In (Czech et al. 2020), it was proposed to use a linear combination of the visits and Q-values to build the move selection policy π. Now, we choose a more conservative approach and only inspect the action with the highest and second highest visits count which we label a_α and a_β respectively. Next, we calculate the difference in their Q-values

$$Q_\Delta(s_0, a_\alpha, a_\beta) = Q(s_0, a_\beta) - Q(s_0, a_\alpha), \tag{13}$$

and if $Q_\Delta(s_0, a_\alpha, a_\beta)$ was found to be > 0, then we boost $\pi(a_\beta, |s_0)$ by $\pi_{\text{corr}}(s_0, a_\alpha, a_\beta)$

$$\pi(a_\beta, |s_0)' = \pi(a_\beta, |s_0) + \pi_{\text{corr}}(s_0, a_\alpha, a_\beta) \; , \tag{14}$$

where

$$\pi_{\text{corr}}(s_0, a_\alpha, a_\beta) = Q_{\text{weight}} Q_\Delta(s_0, a_\alpha, a_\beta) \pi(a_\alpha, |s_0), \tag{15}$$

and re-normalize π afterwards. Q_{weight} acts as an optional weighting parameter which we set to 2.0 because our Q-values are defined to be in $[-1, +1]$.

This technique only involves a small constant overhead and helps to switch to the second candidate move faster, both in tournament conditions as well as when used in the target distribution during RL.

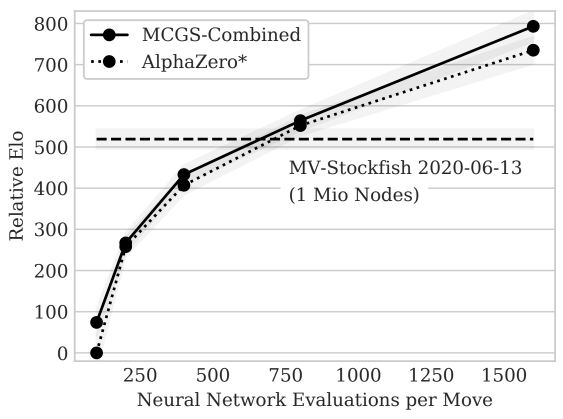

Figure 3: Elo development relative to the number of neural network evaluations in **crazyhouse**.

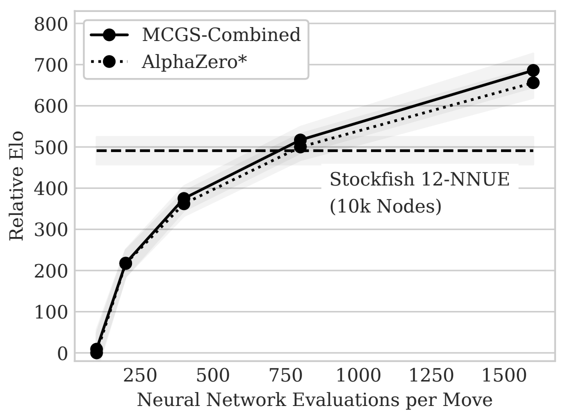

Figure 4: Elo development relative to the number of neural network evaluations in **chess**.

Incorporating Domain Knowledge through Constraints

It is a common advice in chess — both on amateur and master level — to first explore moves that are checks, captures and threats. Hence, we add the constraint to first explore all checking moves before other moves during ϵ-greedy exploration. We again choose a depth according to (11) and follow the so far best known move sequence for expanding a so far unvisited checking move.

The checking moves are being explored according to the ordering of the neural network policy, which makes the exploration of more promising checking moves happen earlier. After all checking moves have been explored, the node is assigned a flag and the remaining unvisited nodes are expanded. After all moves have been expanded, the same procedure as for ϵ-greedy search is followed. As above, we disable backpropagation for all preceding nodes on the trajectory when expanding a check node and choose an ϵ_{check} value of 0.01. In this scenario, we test the expansion of checking moves but it could be extended to other special move types, such as capture or threats as well.

A benefit of this technique is that it provides the guarantee, that all checking at earlier depth are explored quickly, even when the neural network policy fails to acknowledge them, and without influencing the value estimation of its parent nodes.

Empirical Evaluation

Our intention here is to evaluate the benefits of our MCGS and the aforementioned search modifications empirically. In particular, we want to investigate whether each of the contributions boost the performance of *AlphaZero's* planning individually and whether a combination is beneficial.

In our evaluation we use pre-trained convolutional neural network models. For crazyhouse we use a model of the `RISEv2` (Czech et al. 2020) architecture which was first trained on $\approx$0.5 million human games of the lichess.org database and subsequently improved by 436$\pm$30 Elo over

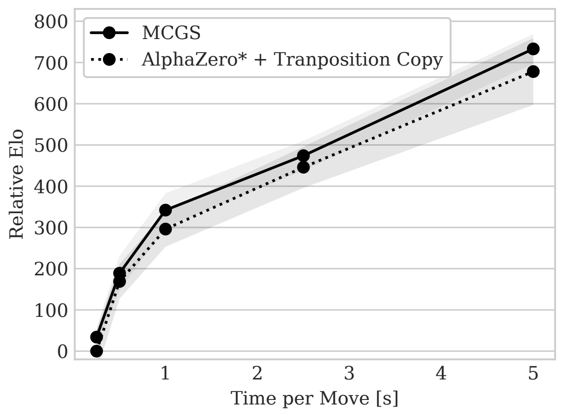

Figure 5: Elo development in **crazyhouse** over time of MCGS compared to MCTS which uses a hash table as a transposition buffer to copy neural network evaluations.

the course of $\approx$2.37 million self-play games [2].

The same network architecture was then employed for chess and trained on the free Kingbase 2019 dataset (Havard 2019) with the same training setup as in (Czech et al. 2020). After convergence the model scored a move validation accuracy of 57.2 % and a validation mean-squared-error of 0.44 for the value loss. This re-implementation is labeled as *AlphaZero** because of not being the original implementation and a different set of hyperparameters values. One of the changes compared to *AlphaZero* is the avoidance of Dirichlet noise by setting ϵ_{Dir} to 0.0 and the usage of Node$_\tau$ which flattens the policy distribution of each node in the search tree. The hardware configuration used in our experiments achieves about 17 000 neural network evaluations per second for the given neural network architecture.

In our first experiment as shown in Figure 5, we compare the scaling behaviour in crazyhouse between our presented MCGS algorithm and a re-implementation of the *AlphaZero*

[2] `https://github.com/QueensGambit/CrazyAra`

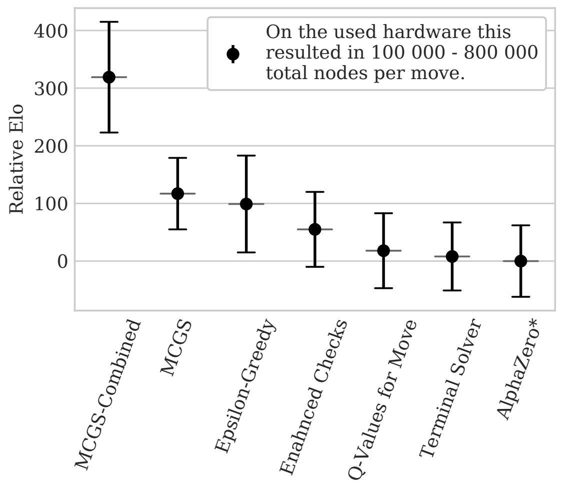

Figure 6: Elo comparison of the proposed search modification in **crazyhouse** using **five seconds per move**.

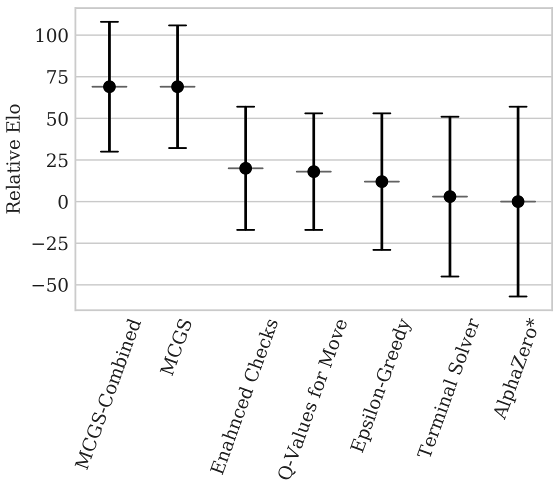

Figure 7: Elo comparison of the proposed search modification in **chess** using **five seconds per move**.

algorithm that also makes use of a transposition table to store neural network evaluations. We observe that MCGS outperformed the transposition look-up table approach across all time controls, demonstrating that MCGS can be implemented efficiently and excels by providing a more expressive information flow along transposition nodes. In particular, it becomes apparent, that the Elo gap between the two algorithms slightly increases over time, suggesting an even better performance in the long run or when executed on stronger hardware.

Next, we investigate the scaling behaviour relative to the number of neural network evaluations in Figure 3 and 4. Again, we can draw a similar conclusion. For a small amount of nodes, the benefit of using all MCGS with all enhancements over *AlphaZero** is relatively small, but increases the longer the search is performed. We state the number of neural network evaluations per move instead of the number of simulations because this provides a better perspective on the actual run-time cost. Terminal visits as well as the backpropagation of correction values $Q_\phi(s_t, a)$ as shown in (10) can be fully executed on CPU during the GPU computation.

To put the results into perspective we also add the performance of *Multi-Variant Stockfish*[3] *2020-06-13* 3 in Figure using one million nodes per move. For chess we use the official *Stockfish 12* release with 10 000 nodes per move as our baseline, which uses a NNUE-network (Yu 2018)[4] as its main evaluation function. The playing strength between the MCGS for crazyhouse and chess greatly differs in strength, however, this is primarily attributed to the model weights and not the search algorithm itself. For evaluating the performance in both chess and crazyhouse we use a list of different starting positions to have more diverse games, fewer draws and in crazyhouse a lower opening advantage for the first player.[2]

[3]`https://github.com/ddugovic/Stockfish`

[4]`nn-82215d0fd0df.nnue`

At last we evaluate each search modification individually as seen in Figure 6 and Figure 7. The amount of nodes per game generally increases over the course of a game because the subtree of the previous search is reused for proceeding searches.

Each individual search modification appears to improve the performance, whereas using MCGS instead of a tree structure yields the greatest benefit. In crazyhouse, MCGS resulted in ≈+110 Elo followed by ϵ-greedy with ≈+100 Elo. Combining all enhancement at once, which we refer to as MCGS-Combined, leads to ≈+310 Elo and greatly surpassed each variant individually. In the case of chess, the impact is not as evident as for crazyhouse but we also recognize an improvement in performance of ≈+69 Elo when using MCGS and all search enhancements at once.

Conclusions

Our experimental results clearly show that using DAGs instead of trees significantly increases the efficiency and performance of the search in *AlphaZero*. Each individual enhancement that we propose gives better results, but the combination exceeds them all and remains stable for large graphs. Together, they boost the performance of *CrazyAra*, the current state-of-the-art engine in crazyhouse. For chess we see less drastic, but still respectable improvements given the models, that were learned through supervised learning.

Our results suggest that MCGS gains in value, the longer the search is executed or on stronger hardware. Moreover, MCGS should be further improved by sharing key trajectories between nodes that are not an exact transposition but similar to each other. Furthermore, one should move beyond the planning setting and explore, how MCGS effects the learning in an RL setting and non-chess environments.

Beyond that, the proposed techniques are in principal applicable outside the *AlphaZero* framework. It is an interesting avenue for future work, to evaluate them in different search frameworks such as UCT or ϵ-greedy search.

Acknowledgements

The authors are grateful for the valuable feedback of our anynomous reviewers. We also thank Matuiss[5] for continuously testing the *CrazyAra* engine and giving constructive feedback. The move generation, terminal and Syzygy-tablebase validation procedures in *CrazyAra-0.8.4* of the crazyhouse and chess environment have been integrated from *Multi-Variant-Stockfish*. We thank Daniel Dugovic, Fabian Fichter, Niklas Fiekas and the entire Stockfish developer community for providing a fast and stable open-source implementation of these algorithms. We also thank Moritz Willig and our other reviewers for proofreading this paper. Finally, we acknowledge the support of the EU project TAILOR (grand agreement 952215) under ICT-48-2020.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47(2-3): 235–256.

Bauer, D.; Patten, T.; and Vincze, M. 2019. Monte Carlo tree search on directed acyclic graphs for object pose verification. In *International Conference on Computer Vision Systems*, 386–396. Springer.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1): 1–43. doi:10.1109/TCIAIG.2012.2186810.

Chen, Y.-C.; Chen, C.-H.; and Lin, S.-S. 2018. Exact-win strategy for overcoming AlphaZero. In *Proceedings of the 2018 International Conference on Computational Intelligence and Intelligent Systems*, 26–31.

Childs, B. E.; Brodeur, J. H.; and Kocsis, L. 2008. Transpositions and move groups in Monte Carlo tree search. In *2008 IEEE Symposium On Computational Intelligence and Games*, 389–395. IEEE.

Choe, J. S. B.; and Kim, J.-K. 2019. Enhancing monte carlo tree search for playing hearthstone. In *2019 IEEE Conference on Games (CoG)*, 1–7. IEEE.

Czech, J.; Willig, M.; Beyer, A.; Kersting, K.; and Fürnkranz, J. 2020. Learning to Play the Chess Variant Crazyhouse Above World Champion Level With Deep Neural Networks and Human Data. *Frontiers in Artificial Intelligence* 3: 24. ISSN 2624-8212. doi:10.3389/frai.2020.00024. URL https://www.frontiersin.org/article/10.3389/frai.2020.00024.

Ecoffet, A.; Huizinga, J.; Lehman, J.; Stanley, K. O.; and Clune, J. 2021. First return, then explore. *Nature* 590(7847): 580–586.

Hassabis, D.; Kumaran, D.; Summerfield, C.; and Botvinick, M. 2017. Neuroscience-inspired artificial intelligence. *Neuron* 95(2): 245–258.

[5]github.com/Matuiss2

Havard, P. 2019. The free KingBase Lite 2019 database. https://archive.org/details/KingBaseLite2019. Accessed: 2020-12-19.

Keller, T. 2015. *Anytime optimal MDP planning with trial-based heuristic tree search*. Ph.D. thesis, University of Freiburg, Freiburg im Breisgau, Germany.

Keller, T.; and Helmert, M. 2013. Trial-based heuristic tree search for finite horizon MDPs. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 23.

Pélissier, A.; Nakamura, A.; and Tabata, K. 2019. Feature selection as monte-carlo search in growing single rooted directed acyclic graph by best leaf identification. In *Proceedings of the 2019 SIAM International Conference on Data Mining*, 450–458. SIAM.

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

Saffidine, A.; Cazenave, T.; and Méhat, J. 2012. UCD: Upper Confidence bound for rooted Directed acyclic graphs. *Knowledge-Based Systems* 34: 26–33.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *nature* 550(7676): 354–359.

Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15(1): 1929–1958.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning* 8(3-4): 279–292.

Winands, M. H.; Björnsson, Y.; and Saito, J.-T. 2008. Monte-Carlo tree search solver. In *International Conference on Computers and Games*, 25–36. Springer.

Yu, N. 2018. NNUE Efficiently Updatable Neural-Network based Evaluation Functions for Computer Shogi. URL https://raw.githubusercontent.com/ynasu87/nnue/master/docs/nnue.pdf.