Know your Enemy: Investigating Monte-Carlo Tree Search with Opponent Models in Pommerman

Jannis Weil*
Technical University of Darmstadt
Darmstadt, Germany
jannis.weil@tu-darmstadt.de

Johannes Czech*
Technical University of Darmstadt
Darmstadt, Germany
johannes.czech@cs.tu-darmstadt.de

Tobias Meuser
Technical University of Darmstadt
Darmstadt, Germany

Kristian Kersting
Technical University of Darmstadt, hessian.ai, DFKI
Darmstadt, Germany

ABSTRACT

In combination with Reinforcement Learning, Monte-Carlo Tree Search has shown to outperform human grandmasters in games such as Chess, Shogi and Go with little to no prior domain knowledge. However, most classical use cases only feature up to two players. Scaling the search to an arbitrary number of players presents a computational challenge, especially if decisions have to be planned over a longer time horizon. In this work, we investigate techniques that transform general-sum multiplayer games into single-player and two-player games that consider other agents to act according to given opponent models. For our evaluation, we focus on the challenging Pommerman environment which involves partial observability, a long time horizon and sparse rewards. In combination with our search methods, we investigate the phenomena of opponent modeling using heuristics and self-play. Overall, we demonstrate the effectiveness of our multiplayer search variants both in a supervised learning and reinforcement learning setting.

KEYWORDS

Multi-agent planning, supervised learning, reinforcement learning

1 INTRODUCTION

Monte-Carlo Tree Search (MCTS) is widely know as a powerful search algorithm for both, single player environments such as Atari, and two player zero-sum games like the game of Go [24]. In environments with more players, MCTS is usually combined with domain-specific heuristics to make search feasible [28].

Multiplayer games with more than two players introduce new challenges to search-based methods. In particular, the size of the search tree explodes when move combinations for multiple players have to be considered [12]. This leads to an exponentially increasing computational complexity of the search, depending on the number of players. Combinations with MCTS [13] lead to improvements across multiple domains, but perform poorly under limited resources due to a shallow search depth [2]. Multiplayer search methods like Parandoid search [27], Best Reply Search [23] and recent extensions [3] improve the search depth by reducing the time spent to simulate opponents and expectedly suboptimal moves.

*These authors contributed equally to this work.

Proc. of the Adaptive and Learning Agents Workshop (ALA 2023), Cruz, Hayes, Wang, Yates (eds.), May 29-30, 2023, London, UK, https://alaworkshop2023.github.io/. 2023.

Additionally, the search can be guided with value estimates [14, 29] and learned value functions [16, 18].

With this work, we focus on the combinatorial aspect of multiplayer games and investigate learning-based MCTS variants that effectively reduce the search space to single- and two-player games. Other players act according to given opponent models, hence drastically reducing the branching factor. We give insights regarding the applicability in a Reinforcement Learning (RL) setting and evaluate our approach in the multiplayer game Pommerman [19].

We provide the following contributions:

- Based on learning-based MCTS, we propose techniques to reduce the search space by transforming competitive n-player games to single-player and two-player games.
- We compare learning from demonstrations, reinforcement learning, and different opponent models in terms of resulting performance and behavior.
- We show that the proposed agent achieves a proficient level of play in the Free For All (FFA) Pommerman environment.

Our code is available at https://github.com/jw3il/PommerLearn. We begin by introducing our approach in a general setting. Next, we go over our experiments in the Pommerlearn environment for both, learning from demonstration data, as well as learning in a reinforcement learning setting. Afterwards we discuss the results and the limitations of our approaches. At last, we present related work and conclude with an outlook for potential future work.

2 APPROACH

When a model of the environment is available, leveraging this knowledge with model-based algorithms comes with several advantages. Our work builds upon MCTS, a general search method that aims to find a sequence of moves leading a player to the expectantly most advantageous states, i.e. states in which the they can win the game. This requires a model of the environment and a way to evaluate states, which could both be provided or learned. By expanding potential future states, the search can make use of additional game knowledge to correct decisions where a suboptimal agent alone would fail. This allows to filter potential dangers instead of having to face them. Additionally, search methods usually return a principal variation, which is the sequence of future moves that are considered best under the current knowledge. By iterating over this sequence, it is possible to give a more throughout explanation of the planned behavior of the agent.

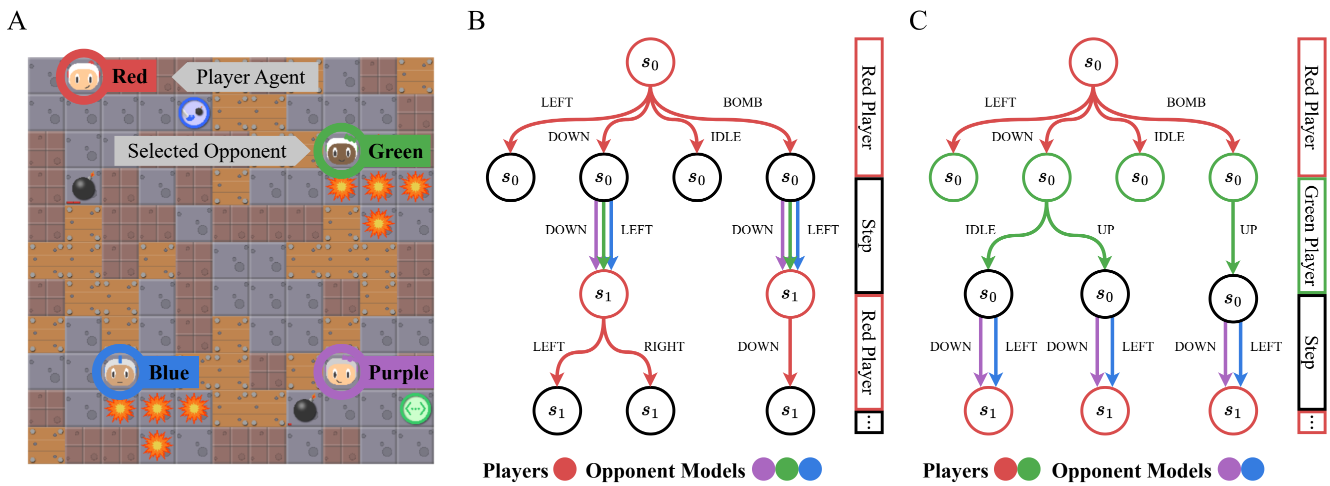

Figure 1: Exemplary search graphs of Single-Player Search (B) and Two-Player Search (C) next to a Pommerman board (A). The Single-Player Search allows a deeper search with fully heuristic-based play for all opponents, whereas the Two-Player Search allows a full exploration of a selected opponent at each step with the downside of achieving a lower search depth.

Search methods require a strategy for selecting and expanding nodes in a search tree. In our case, this is provided by a neural network model that predicts value and policy distributions for each state, as we will detail later. The search focuses on potential future states that appear to be promising, while also exploring other paths. We use a variant of the Predictor Upper Confidence Bounds (PUCT) algorithm [22] to select and expand new nodes. In particular, we refer to the algorithm adjusted by Silver et al. [26]:

$$a_t = \operatorname{argmax}_a \left(Q(s_t, a) + U(s_t, a) \right) , \tag{1}$$

$$\text{where} \quad U(s_t, a) = c_{\text{puct}} P(s_t, a) \frac{\sqrt{\sum_b N(s_t, b)}}{1 + N(s_t, a)} . \tag{2}$$

Here, a_t refers to the selected action at time step t, and $Q(s_t, a)$ is the action value for action a of state s_t. The action values of a node are updated by calculating a simple moving average of all backpropagated value estimates of its subtree. The term $U(s_t, a)$ describes the utility function for a particular action. It is given by the product of its policy estimate $P(s_t, a)$ and the total number of visits of its parent divided by the number of visits $N(s_t, a)$ of the selected action. This prioritizes actions that have a higher policy estimate or were chosen less frequently. The denominator $1 + N(s_t, a)$ is used to avoid division by zero, and so that the nodes do not have to be fully expanded over each action. The scalar c_{puct} is a weighting parameter which controls the amount of exploration compared to the greedy action selection of choosing the highest Q-value.

Extensions of MCTS to games with more than two players come with conceptual and practical challenges. In particular, the search space grows exponentially with the number of players if all of them are considered in the search.

To address this issue, we propose two simple yet effective methods that reduce multiplayer games to single and two-player games. They allow for the application of AlphaZero-like frameworks without major adjustments. We describe the methods in the following sections and provide a visualization with Fig. 1.

2.1 Single-Player Monte-Carlo Tree Search

A straightforward approach for simplifying the search is to transform the multi-player environment into a single-player environment by treating the opponents as part of the environment. Instead of modifying the environment's dynamics, we simplify the search space by using deterministic opponent models for other players. This also builds the basis of our second approach, which we will introduce in the next section. Instead of searching through all actions of all players, we limit the search to our player agent. In Fig. 1 (B), this is the red player. The search tree is expanded solely using the actions of this player. To execute a step in the deterministic environment, we then gather actions for other players with their deterministic opponent models. With all actions, the environment advances from state s_0 to the next state s_1. Using this method, an n-player game effectively reduces to a single-player game.

The quality of the resulting policy depends on the given opponent model. If the opponent model differs from the actual behavior of the opponents, the paths that are explored during search can get highly inaccurate and diverge from potential future trajectories in the real environment. Convergence guarantees towards optimal behavior are lost and a higher search depth could even lead to deteriorations of the resulting policy.

Despite these unfavorable preconditions, our hypothesis is that this approach can allow to assess the current situation in order to make good decisions with respect to immediate dangers and the near future. The better the opponent model fits the behavior of our opponents, the better we can exploit their behavior. As we simulate a single agent, we can perform many simulation steps that, although being inaccurate, could help to estimate the value of the available actions. If one would use an optimal player as the opponent model, our agent would plan how to act in the worst-case scenario, irrespective of the actual behavior of the opponents.

Given an action space $\mathcal{A}$, the maximum branching factor per step is $|\mathcal{A}|$ as we only expand moves of one agent.

Algorithm 1: Single-player MCTS for multiplayer games. Shown is a single tree search update iteration (simulation).

```
Function SP_MCTS(tree, playerID, opponentIDs):
    node, state ← SelectLeafNode(tree)
    action ← SelectAction(node)
    actions[playerID] ← action
    for idx ∈ opponentIDs do
        actions[idx] ← OpponentModel(idx, state)
    state′ ← EnvironmentStep(state, actions)
    result ← Evaluate(state′, playerID)
    tree′, node′ ← ExpandTree(tree, node, action, result)
    return BackpropagateSP(tree′, node′)
```

This search method is summarized in Alg. 1. The function SelectLeafNode corresponds to the node selection phase in MCTS and selects a leaf node by following Eq. (1), SelectAction then selects a new action in this leaf. After gathering the remaining actions with the given opponent models, performing a step in the environment yields a new state. This state is evaluated from the perspective of the player agent and the results are stored in a new node which is appended to the tree. Finally, the value from the new node is backpropagated without depth-wise negation using BackpropagateSP, returning the updated tree.

2.2 Two-Player Monte-Carlo Tree Search

The main limitation of our single-player search is that the play behavior of our opponents remains deterministic, alternative moves are not considered, and it cannot converge to an optimal strategy during the search if there is a discrepancy between the opponent model and the actual opponent behavior. To overcome these limitations to some extend, we propose an approach which we call two-player search. This approach expands our single-player search by exploring the moves of a selected opponent in each step, e.g. the green agent in Fig. 1 (C). Instead of following the deterministic opponent model, the move nodes for this opponent can be fully expanded. The selected opponent makes use of the same prior policy for move selection as our agent. All remaining opponents perform actions according to their given models. Note that the selected opponent can change across steps in the simulation. Ideally, one would select the opponent that, when allowed to deviate from the given opponent model, results in the most reduction in our agent's estimated value. This can be seen as an instance of Best Reply Search (BRS) [23] where the opponent with the best reply is assumed to be known. For simplicity, we choose the closest agent. While BRS skips moves of opponents that are not selected, BRS+ [6] uses move orderings to select valid moves. We use opponent models to advance other opponents during the search. In the example in Fig. 1 (C), our approach additionally expands the actions of the green player. Like in vanilla MCTS, the values of the green player are negated and then backpropagated to the red player. The other opponents are seen as a part of the environment during this step.

The two-player search expands moves for selected opponents, thus leading to a higher branching factor compared to the single-player search. Given an action space $\mathcal{A}$, the maximum branching factor per step is now $|\mathcal{A}|^2$. This is magnitudes smaller than the maximum branching factor of complete enumeration with $|\mathcal{A}|^n$ for n agents and of BRS with $(n-1)|\mathcal{A}|^2$.

Algorithm 2: Two-player MCTS for multiplayer games. Shown is a single tree search update iteration (simulation).

```
Function TP_MCTS(tree, playerID, opponentIDs):
    node, state ← SelectLeafNode(tree)
    agentID ← GetActiveAgent(node)
    action ← SelectAction(node)
    if agentID = playerID then              // player node
        result ← Evaluate(state, playerID)
    else                          // opponent node and step
        actions[agentID] ← action
        actions[playerID] ← GetLastAction(node)
        for idx ∈ opponentIDs \ {agentID} do
            actions[idx] ← OpponentModel(idx, state)
        state′ ← EnvironmentStep(state, actions)
        result ← Evaluate(state′, agentID)
    tree′, node′ ← ExpandTree(tree, node, action, result)
    return Backpropagate(tree′, node′)
```

The pseudocode is shown in Alg. 2. We alternately select a new action for the player agent and a selected opponent and expand the tree accordingly. After selecting both actions, the opponent models fill in the remaining actions to perform a step in the environment. Note that this uses regular backpropagation with negated values.

2.3 Combination with Learned Models

Based on the idea of AlphaZero [26], we leverage an agent model $f_\theta(o) = (\mathbf{p}, v)$ to predict move probabilities $\mathbf{p}$ and a value v for a given observation o. These predictions are used to guide the previously described search approaches. In the PUCT formula (see Eq. 1), $P(s_t, a)$ evaluates to $\mathbf{p}$ and v is used to update $Q(s_t, a)$ upon expanding non-terminal nodes. The loss is defined as

$$l = \alpha(z - v)^2 - (1 - \alpha)\pi^\top \log \mathbf{p} \,, \tag{3}$$

where z is the target value and π the move probability according to the search. The total loss consists of both the value loss, that is given as a mean squared error, and the policy loss, that is formulated as a cross-entropy loss. Hyperparameter α weights the value loss, Silver et al. [26] suggested using a low weight to reduce the chance of overfitting to the value target. We iteratively update the model with the AdamW optimizer [11].

3 EXPERIMENTS IN POMMERMAN

Pommerman [19] is a multi-agent environment inspired by the video game series *Bomberman*. Up to four bomber agents move across a discrete grid-world and try to defeat their opponents by placing bombs. In the FFA mode, each agent plays on their own and observes the the whole board except for hidden power ups. In the *team* and *radio* modes, there are two teams of two agents each. Agents can only observe their local surroundings up to a distance of 4 blocks from their current position, horizontally or vertically. In the radio mode, agents can additionally use a discrete communication channel and send six bits per step. The FFA mode

Table 1: Overview of used agents.

Abbreviation	Description
SimplePy	SimpleAgent from the Python environment.
SimpleCpp	SimpleUnbiasedAgent from pomcpp2.
RawNet	Chooses the action with the highest Q-value of the player model.
SP-MCTS	Our approach with the single-player search.
TP-MCTS	Our approach with the two-player search.

has been used in a preliminary competition in 2018 [19], the teams mode at NeurIPS 2018 [20] and the radio mode at NeurIPS 2019[1].

The Pommerman environment is very challenging, mainly because it is a multiplayer game, its long time horizon of up to 800 steps and partial observability. With four players and $|\mathcal{A}| = 6$ actions, exhaustively exploring the search tree for 10 steps in order to see a newly placed bomb explode would require evaluating around $(6^4)^{10} \approx 1.34e^{31}$ states. Given the environment's time limit of 100 milliseconds per move, there is a need for more efficient solutions.

3.1 Training Setup

We implement our approach on top of CrazyAra [4], an AlphaZero-like MCTS framework that includes several extensions. Our agent model f_θ uses a *RISEv2 mobile* architecture [5] adapted for the game Pommerman. The input of the model is of board size 11×11 with 23 feature channels that encode the agent's observation. Further details are provided in our repository. The learning target z is the outcome of an episode and either win (1), draw (0) or loss (−1). Custom intermediate rewards and discounting are not used.

Each training iteration is performed in a supervised manner on given datasets according to the loss in Eq. (3) with $\alpha = 0.1$ to avoid overfitting to the value target. We perform data augmentation to mirror and rotate all observations jointly with the targets to improve the sample efficiency. Depending on the experiment, the datasets either originate from expert demonstrations or from samples generated by our search approaches.

The official Pommerman environment [19] is implemented in Python and provides baseline agents called SimpleAgent. Our approach is implemented in C++ and makes use of a faster reimplementation of the Pommerman environment [30]. This includes an agent called SimpleUnbiasedAgent that improves upon the provided C++ SimpleAgent and reduces the decision bias depending on the agent's id.[2] Most of the results presented in the following sections use this reimplementation and the FFA mode, but we conclude with preliminary results in the Python environment. An overview of the considered agents is presented in Tab. 1.

3.2 Learning from Demonstrations

To investigate the effectiveness of our search approaches, we first study their combination with learning from demonstrations. We generate a data set with one million samples of SimpleCpp agents playing FFA games with random start conditions for up to 800 steps.

[1]https://nips.cc/Conferences/2019/CompetitionTrack.
[2]We found that the C++ SimpleAgent behaves differently depending on its agent id and start position, resulting in varying average win rates.

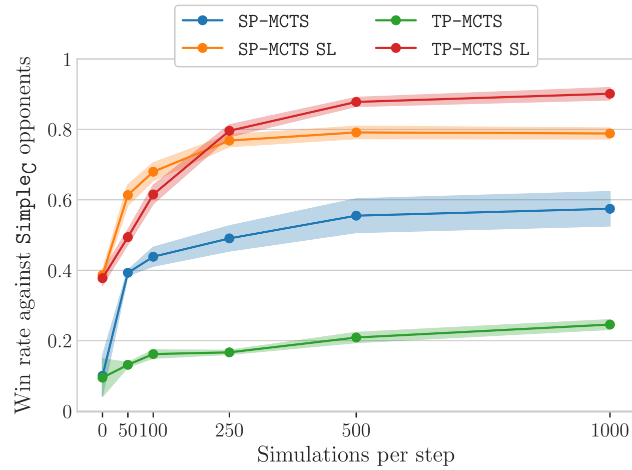

Figure 2: TP-MCTS SL outperforms SP-MCTS SL for higher number of simulations. When random initialized models are used, SP-MCTS has consistently higher win rates than TP-MCTS. Shown are the win rates of SP-MCTS and TP-MCTS against SimpleCpp in the FFA mode. The suffix SL indicates that the search uses a model trained on demonstrations. The standard deviation of five runs is highlighted in the shaded area.

This includes samples from the perspective of each player, i.e. we collect four trajectories per episode. The model is trained using our loss from Eq. (3) with Supervised Learning (SL), where the target policy equals the actions chosen by the agents. The resulting model is used as the player agent in conjunction with the search methods SP-MCTS and TP-MCTS. For these experiments, we set the opponent models to SimpleCpp with random seeds. Thus, the search cannot foreshadow the exact moves that will be selected by the actual opponents, but captures their overall behavior.

Fig. 2 shows the win rate of our approaches over 1000 games against SimpleCpp opponents for increasing simulations per step. For zero simulations, we use the respective RawNet agent that chooses an action based on the maximum probability of the root nodes's policy distribution without any look-ahead. The results are averaged over five models trained on the same data set with different seeds. Note that with four agents in the FFA mode, a win rate of 25% indicates equal performance if there are no draws. We include the results for randomly initialized models as a baseline.

We can see in Fig. 2 that for the randomly initialized models, SP-MCTS highlighted in blue greatly outperforms TP-MCTS highlighted in green. This is because TP-MCTS uses the given model to guide the search of the closest opponent. Our agent tries to exploit the mistakes of its opponents. If the opponent model is poor, the agent gets overconfident in taking bad actions and the search results do not transfer well to the real environment.

This is consistent with the comparably good results of TP-MCTS SL when the expert model is used. While SP-MCTS SL outperforms TP-MCTS SL for a low number of simulations, TP-MCTS SL achieves higher win rates for 250, 500 and 1000 simulations per step. The win rate for zero simulation steps of the learned model is greater than 25%, which indicates that its combination with action filtering

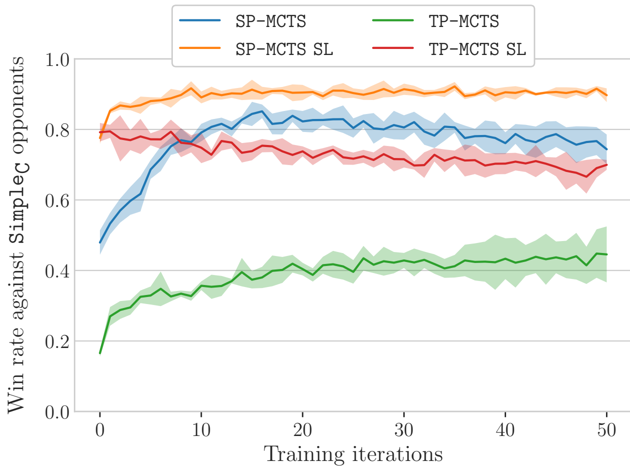

Figure 3: In reinforcement learning, both SP-MCTS versions appear to perform better than TP-MCTS. Shown are the win rates of RL-based SP-MCTS and TP-MCTS against SimpleCpp in the FFA mode over 50 training iterations with 250 simulations. The configurations with suffix SL are initialized with the model trained on demonstrations. The standard deviation of five runs is highlighted in the shaded area.

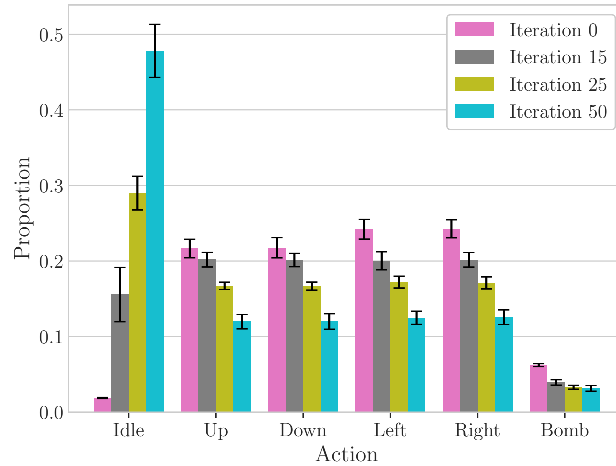

Figure 4: SP-MCTS SL becomes more passive during reinforcement learning over time, reflected by the higher use of idle actions. Shown is the action distribution of SP-MCTS SL during training with RL at iterations 0, 15, 25 and 50. The standard deviation of five runs is highlighted by the error bars.

already performs better than SimpleCpp. With a high number of simulations, TP-MCTS SL can reach a sufficient search depth and benefit from an increased exploration of the opponent's actions.

To summarize, the result for the model initialized with expert demonstrations are promising and we see that both search approaches greatly improve the performance of a randomly initialized model. We now investigate whether these models can be improved by iteratively training on samples generated by the search.

3.3 Optimization with Reinforcement Learning

As the next step, we aim to improve the models from the previous section with RL. We simulate FFA games against SimpleCpp opponents for 100 000 steps and use the search's results as the learning target for the policy and value functions. The resulting model is then used in the next iteration and the process is repeated. As before, all agent and opponent models are set to SimpleCpp. We train our agent for 50 iterations with 250 simulations per step. This setting has been chosen as a trade-off between required simulation time and win rate based on the previous experiments.

During our experiments, we noticed that the results highly depend on the amount of noise introduced by exploration within the search. With the regular policy target π, the agents get stuck in local optima with low win rates after around 15 iterations. In the following, we focus on a configuration with a modified policy target $\pi' = 0.5 \cdot \pi + 0.5 \cdot \mathbb{1}\{\pi = \max \pi\}$, as we found the corresponding results to be more insightful. This reduces the noise introduced by the search and in turn increases the probability of choosing the best action with the highest visit count by 50%.

The results in Fig. 3 are averaged over five runs using the models from the previous section. We can see that the win rate of SP-MCTS and TP-MCTS increases, suggesting improvements of the corresponding models. However, the win rate of SP-MCTS reaches its peak at around 15 iterations and starts to slowly decline afterwards. While the win rate of SP-MCTS SL slightly increases from 80% to 90%, the win rate of TP-MCTS SL decreases over time. Investigating the resulting policies reveals that the agents learn to play passively with RL, i.e. they wait for their opponents and evade bombs when necessary. This strategy is unexpected but expedient, as SimpleCpp opponents are suboptimal and often put themselves in unfavorable situations.

This is particularly visible in the SP-MCTS SL configuration with a win rate of around 90% after training with RL. We show our agent's action distribution for selected training iterations in Fig. 4. For iteration 0, this is the action distribution of the original SL models. It can be seen that the model gradually shifts from an initially active policy with few idle actions to a policy that predominantly idles. While the results show that this is a successful strategy against SimpleCpp, the passive behavior will fail against better opponents.

The decreasing win rate of TP-MCTS SL is consistent with these findings. While SP-MCTS SL assumes the opponents to behave like SimpleCpp, TP-MCTS SL uses its own policy to expand the moves of the closest opponent. By selecting idle with a higher probability, the opponent model diverges more and more from the actual opponent playing behavior and the win rate of this approach decreases.

3.4 Learned Opponent Models

In the previous sections, we investigated our search methods in combination with supervised and reinforcement learning. However, we still used the heuristic SimpleCpp as the opponent model during planning. As SimpleCpp is clearly suboptimal, this may lead to problems when applying the agent against opponents with strategies that differ significantly. Consequently, we explore the usage of

Table 2: Win rate, tie rate, search depth, search runtime per step and environment steps for our approaches with 1000 simulations and different opponent models against SimpleCpp opponents for 1000 games. All results are averaged over five models.

Model	Method	Opponent Model	Win Rate	Tie Rate	Search Depth	Search Time [ms]	Environment Steps
SL	SP-MCTS	SimpleCpp	0.78 ± 0.03	0.07 ± 0.02	17.91 ± 7.60	35.30 ± 7.57	188.99 ± 3.20
		RawNet	0.76 ± 0.03	0.10 ± 0.02	21.46 ± 22.09	266.62 ± 138.13	191.02 ± 2.91
	TP-MCTS	SimpleCpp	**0.91 ± 0.01**	0.06 ± 0.01	11.57 ± 6.05	36.89 ± 7.52	246.40 ± 5.93
		RawNet	**0.92 ± 0.01**	0.06 ± 0.01	12.14 ± 7.17	164.61 ± 64.46	254.79 ± 11.26
SRL	SP-MCTS	SimpleCpp	**0.94 ± 0.01**	0.02 ± 0.01	21.39 ± 7.45	37.34 ± 6.74	275.23 ± 9.40
		RawNet	**0.90 ± 0.01**	0.04 ± 0.01	25.52 ± 10.37	315.74 ± 139.75	288.47 ± 10.26
RL	SP-MCTS	SimpleCpp	0.82 ± 0.01	0.07 ± 0.00	21.98 ± 8.65	36.49 ± 6.52	331.99 ± 7.38
		RawNet	0.71 ± 0.03	0.08 ± 0.01	28.30 ± 11.73	309.40 ± 140.20	316.81 ± 9.46

RawNet opponent models within this section. As the combination of our model learned from demonstrations with our simple action filter is apparently better than SimpleCpp, agents using RawNet opponent models should be capable of adapting to better players.

Tab. 2 shows the results of our search approaches for SimpleCpp and RawNet opponent models against SimpleCpp opponents. The search uses 1000 simulations to be comparable to the results from Fig. 2. We focus on SP-MCTS in the RL setting due to the weak performance of TP-MCTS and show the results for the models initialized from zero (RL) and the ones initialized with SL and refined with RL (SRL). For both, we use the models after 15 training iterations due to the peak in Fig. 3. All experiments were performed for each of the five respective models and we report the mean results. Ties include episodes that are not done.

For the SL models, we can see that the win, tie rates and environment steps are nearly unaffected when using RawNet instead of SimpleCpp for both of our approaches. However, it is noticeable that the search depth increases slightly and the search time increases drastically. We hypothesize that the increase in search depth is caused by the better opponent behavior, i.e. episodes within the search do not end as quickly as RawNet is a stronger opponent. However, as the actual opponents are still SimpleCpp, this is not reflected in the real environment, as visible in similar numbers of environment steps. The increase in search time can be explained by our prototypical implementation of the RawNet opponent models. While the model inference for SP-MCTS and TP-MCTS is executed in batches, we currently use batch size one for RawNet opponent models within the search. This drastically increases the time to evaluate the opponent models per step. With an halved search depth in TP-MCTS, the search time also decreases greatly as there are fewer inference calls of the opponent model.

For the SRL and RL models, we notice that the win rate slightly decreases when using RawNet opponent models. However, the SRL model yields higher win rates than SP-MCTS with the SL model and is comparable to TP-MCTS with the SL model. The win rate of the RL model without training on demonstrations is similar to the initial SL model for SP-MCTS. For the SRL and RL models, there is high increase in environment steps compared to the SL models. This indicates that the agents play more passively.

We conclude that in most cases, the RawNet opponent model has a neglectable effect on the win rate against opponents that were seen in training. With a more efficient implementation, it could be a good alternative to hand-crafted heuristics. The best agents are TP-MCTS with the SL model and SP-MCTS with the SRL model.

Table 3: Win rate and tie rate of our approaches against SimplePy opponents in the official environment for 100 games. We use 1000 simulations for SimpleCpp and 250 simulations for RawNet. All results are averaged over five models.

Model	Method	Opponent Model	Win Rate	Tie Rate
SL	SP-MCTS	SimpleCpp	**0.78 ± 0.04**	0.12 ± 0.02
		RawNet	**0.67 ± 0.06**	0.25 ± 0.04
	TP-MCTS	SimpleCpp	0.66 ± 0.05	0.32 ± 0.05
		RawNet	0.61 ± 0.06	0.34 ± 0.06
SRL	SP-MCTS	SimpleCpp	0.65 ± 0.09	0.30 ± 0.09
		RawNet	0.63 ± 0.05	0.32 ± 0.04
RL	SP-MCTS	SimpleCpp	**0.74 ± 0.03**	0.22 ± 0.03
		RawNet	**0.72 ± 0.04**	0.23 ± 0.03

3.5 Evaluation in the Official Environment

Finally, we evaluate our approach in the official Python environment [19] to investigate whether our previous results transfer to the Python environment and the SimplePy opponent. The starting positions are randomized to reduce their influence on the results. As the previous section showed a high search time when using RawNet, we restrict the number of simulations to 250 in this case to stay below the official time constraint of 100 ms.

The results against SimplePy agents in the FFA mode are shown in Tab. 3. In contrast to our previous evaluation against SimpleCpp in Tab. 2, SP-MCTS outperforms TP-MCTS by a noticeable margin for the SL models. While the results of SP-MCTS are similar to our previous evaluation, the win rate of TP-MCTS strongly decreases. A potential reason for that could be the difference between the actual opponent behavior and the one considered during planning. The search might assume that opponents play too well, resulting in an overly defensive play style. This is also indicated by the high reduction in the win rate when using the RawNet opponent model in SP-MCTS with the SL models. Another indicator for the defensive playing style is the high tie rate and the high similarity of the results to the SRL models. Interestingly, the RL models outperform the SRL models in this setup. It could be that they generalize better against other opponents because they were not trained on demonstrations.

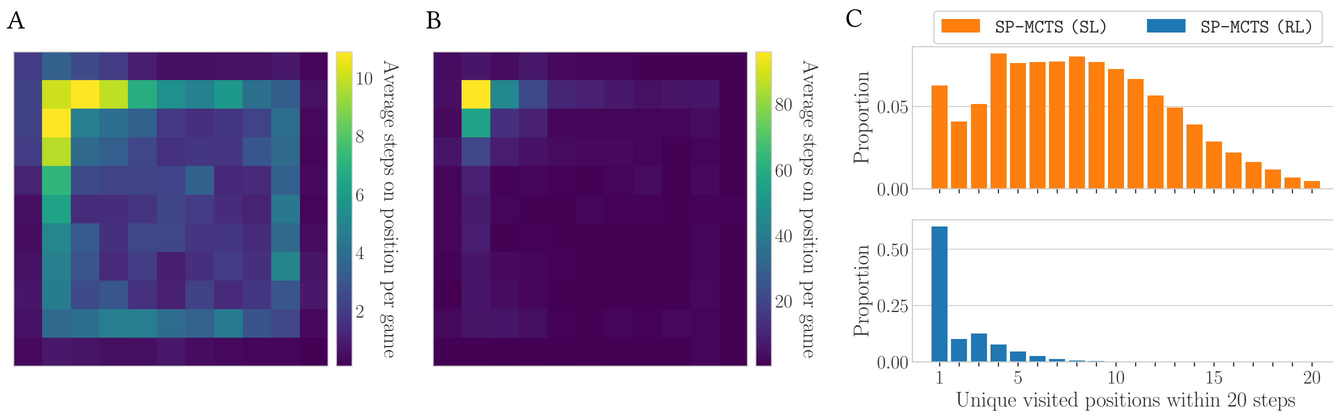

Figure 5: Visited board positions of the SP-MCTS SL (A) and RL agents (B) against SimplePy in the official environment. Subfigure (C) complements this with the number of unique positions visited within 20 steps. While the SP-MCTS SL agents show active movement behavior, the agents trained with RL are very passive. The results are averaged over 5 models with 100 games each.

We notice that the tie rates of all agents are very high, especially for the agents with lower win rates. Most of these ties are caused by episodes that do not terminate within the environment's limit of 800 steps. In turn, the average number of environment steps per episode also increases greatly to up to 490 ± 230 steps for SP-MCTS with the RL model. We omit the steps in the table, their overall trend for the individual models and methods is similar to the previous results. Despite the reduced win rates, our agents still loose very few games due to a defensive play style.

Recent related work with learning-based MCTS and reward shaping reports a win rate of around 0.7 against SimplePy opponents in the FFA mode [32]. Our approaches reach competitive win rates without reward shaping. Additionally, the RL agent was trained from scratch and did not use learning from demonstrations.

We show further details regarding the movement behavior of the SP-MCTS SL and RL agents with SimpleCpp opponent models in Fig. 5. Subfigures (A) and (B) show the average steps on the individual board positions per game. For the visualizations, we rotated the board according to the agents' starting positions such that they always start at the upper left corner. This allows us to see how much the agents explore the map, irrespective of their starting position. In Fig. 5 (A), we can see that the SP-MCTS SL agent actively explores the map while avoiding the border, except for the tiles close to its starting position. This is reasonable, as agents at the border have fewer options for evasion. The noticeable ring across the map at distance one to the border is due to the randomization of the map. Only destructible objects and passages are placed at these positions, ensuring that the agents can reach each other. In Fig. 5 (B) and (C), we can see that the SP-MCTS RL agents stay very close to their starting positions and rarely move across the map, confirming that they develop a very defensive playing style.

4 DISCUSSION

Next, we discuss the insights and limitations of our approach. One major insight is that in Pommerman, focusing on the win rate alone is not enough. While a high win rate is an indicator for subjectively good agent behavior, further analysis of the behavior of the agents is required to assess the quality of their policies.

We have shown that no custom reward shaping is necessary to significantly improve agent models with our search methods. Our approach SP-MCTS can reach proficient level of play in the FFA environment and even compensate for a bad model. TP-MCTS can outperform SP-MCTS with a higher number of simulations, but only when a good model is given. Our evaluation with SimplePy opponents suggest that there might not be a single opponent model that allows the agents to perform well in all cases. If the real opponents show suboptimal behavior that is not captured by the opponent model, our approaches become overly defensive. One limitation of our experiments is that we only considered deterministic opponent models and models trained on SimpleCpp agents. Instead, one could collect samples from the best available agents and train models to imitate their behavior. We expect that combining these models with our search approaches could further increase their playing strength. Stochastic opponent models could be considered by expanding multiple different opponent actions into individual nodes or merging different opponent trajectories into a single node with the expected or worst-case behavior. This would greatly increase the sample complexity, but could reduce wrongful exploitation when facing different opponents and make the search more applicable to realistic scenarios. Another direction for future work lies in the way opponents are selected by TP-MCTS. In our case, we always expand the actions of the closest opponent. Extensions could predict the most dangerous opponent or selectively expand opponents based on a computational budged. Finally, we think that combining more computationally expensive search methods [3, 6] with learned opponent models poses an interesting direction for future research.

While RL improved the win rate of some configurations, the resulting behavior did subjectively deteriorate. In particular, the agents trained via RL were predominantly passive. Providing a fine-grained reward signal might be necessary to learn the desired behavior, at the cost of introducing additional bias. Another interesting aspect of RL would be to investigate pure self-play, e.g. having four learning SP-MCTS agents play against each other. Preliminary experiments not discussed in this paper suggest that self-play agents also develop a passive playing style, further strengthening the need for intermediate rewards when using RL. A different approach to stabilize training in a self-play setting could be to anticipate the learning of other agents in the environment [7]. One problem that emerges without self-play is that high win rates lead to unbalanced data sets. We experimented with resampling techniques to draw samples for each value and action target with equal probability, but this did not lead to noticeable improvements. To avoid overfitting on specific opponents, population-based approaches with different opponents would also be worth investigating.

Lastly, we focused on the FFA mode. Extensions of our approach to the team and radio mode would be interesting. Initial results show that our agent performs well against a team of `SimplePy` opponents, but struggles against docker agents from previous challenges. To deal with the partial observability in form of the now limited view, agent models leveraging recurrent neural networks and learned communication can be explored. We think that including bomb kicks in the demonstration data set, e.g. by considering samples from different agents, could greatly help our agents to react appropriately when facing these opponents. It would also be interesting to combine our approaches with learned environment models, especially to avoid hand-crafting environment dynamics that can handle the limited view in the team and radio modes.

5 RELATED WORK

The combination of RL and tree search has been extensively studied in the domain of games, in particular board games. One breakthrough in this field is the work by Anthony et al. [1], which applied tree search and reinforcement learning to learn the game of Hex. This approach was later followed and popularized in the AlphaZero [25] algorithm by learning the games Go, Shogi and Chess from zero knowledge. AlphaZero has then been re-implemented and extended for Go [31] and multiple chess variants [4]. Later works in the form of MuZero [24] emphasized the environment model by learning a model that is used for planning rather than relying on the actual environment itself.

Planning approaches using tree search have also been successfully applied in multiplayer games with more than two players [12, 13]. However, they can suffer from a shallow search depth [2] in practice due to the high combinatorial complexity. Many approaches increase the search depth by reducing the time spent to simulate opponents and expectedly suboptimal moves [3, 23, 27]. In addition to simplifying the search tree, integrating domain knowledge in the form of value heuristics into MCTS has shown to greatly improve performance in multiplayer games [14, 29]. Petosa and Balch [18] extend the idea of AlphaZero and apply learned value estimation in multiplayer games, but iterate over all players during search. On contrast, Ozair et al. [16] consider opponents to be part of the environment's dynamics and learn a latent variable to sample state transitions for MCTS that include the opponent's moves. While they only considered games with up to two players in their evaluation, the idea should be generally applicable.

Previous work in Pommerman [19] ranges from learning- and planning-based approaches to the combination of both. Due to the sparse reward and long time horizon, approaches leveraging model-free RL struggle to beat simple heuristics without further modifications of the environment or training procedure [9]. Resnick et al. [21] suggest to start training close to terminal states, Peng et al. [17] use pathfinding-based actions instead of direct movement and Gao et al. [8] employ reward shaping, action filtering and curriculum learning. However, agents using model-free RL have shown inferior performance compared to planning-based approaches. Their main limitations are the time constraints for decision making and the high branching factor. The winners of the NeurIPS 2018 competition combine planning with deterministic and pessimistic rollouts to increase the search depth [15]. Rollouts with more than ten steps allow the agents to account for explosions of recently placed bombs during planning. The second-placed agent uses minimax search with an average search depth of only two steps [20], an extension of this agent won the subsequent competition held at NeurIPS 2019. Learning and planning can also be combined. For example, Kartal et al. [10] use model-free RL but initialize their agent with imitation learning on samples generated by shallow MCTS with random agents. Yang et al. combine MCTS with a learned model, they initialize their agent with imitation learning and employ reward shaping and sophisticated action filtering heuristics during search [32].

We observe that, although they reach proficient levels of play, recent planning-based approaches either suffer from a shallow search depth or introduce high bias through search heuristics and reward shaping. In this paper, we explored the feasibility of learning-based MCTS in the Pommerman environment with opponent models, given only an environment model, a sparse reward signal at the end of the episode, and demonstrations from other agents.

6 CONCLUSION

With this work, we proposed two methods based on MCTS that make use of deterministic opponent models and reduce competitive multiplayer games to single- and two-player games. This greatly reduces the complexity of the search space and makes MCTS applicable to complex environments with time constraints. We evaluated our approach in the game Pommerman without custom reward shaping. We found that both methods load to high improvements in terms of win rate against baseline agent heuristics, both when using an uninitialized model and a model trained on demonstrations. Our two-player search outperforms the single-player search, but requires more simulations and a good initial model. While the application of RL based on the samples generated by the search leads to improved win rates in most cases, we found that the agents develop a passive playing style. We think that intermediate rewards might be necessary to learn a more active policy in a RL setup.

Future work could investigate how our approaches perform if demonstrations from better agents are used to train the initial models. To further explore the RL setup, the next step would be to integrate reward shaping. It would also be interesting to expand upon the opponent selection in our two-player search, e.g. by predicting the most dangerous opponent in each step. Another promising direction would be to consider MCTS with stochastic opponent models. To extend our approach to the team and radio modes, the combination of MCTS with recurrent neural networks and learned communication between agents could be explored further.

ACKNOWLEDGMENTS

The authors thank Jonas Ringsdorf for assisting with the implementation and evaluation, Adrian Alic for the development of pomcpp, and Quentin Delfosse as well as Jannis Blüml for valuable feedback. This work benefited from the Hessian Ministry of Science and the Arts (HMWK) project 'The Third Wave of AI' and has been co-funded by the German Research Foundation (DFG) in the Collaborative Research Center (CRC) 1053 MAKI.

REFERENCES

[1] Thomas Anthony, Zheng Tian, and David Barber. 2017. Thinking Fast and Slow with Deep Learning and Tree Search. In *Advances in Neural Information Processing Systems 30*. 5360–5370.

[2] Hendrik Baier and Michael Kaisers. 2020. Guiding Multiplayer MCTS by Focusing on Yourself. In *IEEE Conference on Games*. 550–557.

[3] Hendrik Baier and Michael Kaisers. 2020. Opponent-Pruning Paranoid Search. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*. ACM, 10:1–10:7.

[4] Johannes Czech, Patrick Korus, and Kristian Kersting. 2021. Improving AlphaZero Using Monte-Carlo Graph Search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 31. 103–111.

[5] Johannes Czech, Moritz Willig, Alena Beyer, Kristian Kersting, and Johannes Fürnkranz. 2020. Learning to Play the Chess Variant Crazyhouse Above World Champion Level With Deep Neural Networks and Human Data. *Frontiers in Artificial Intelligence* 3 (2020).

[6] Markus Esser, Michael Gras, Mark H. M. Winands, Maarten P. D. Schadd, and Marc Lanctot. 2013. Improving Best-Reply Search. In *Proceedings of the 8th International Conference on Computers and Games (Lecture Notes in Computer Science, Vol. 8427)*. Springer, 125–137.

[7] Jakob N. Foerster, Richard Y. Chen, Maruan Al-Shedivat, Shimon Whiteson, Pieter Abbeel, and Igor Mordatch. 2018. Learning with Opponent-Learning Awareness. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. 122–130.

[8] Chao Gao, Pablo Hernandez-Leal, Bilal Kartal, and Matthew E. Taylor. 2019. Skynet: A Top Deep RL Agent in the Inaugural Pommerman Team Competition. (2019). arXiv:1905.01360 [cs.MA]

[9] Chao Gao, Bilal Kartal, Pablo Hernandez-Leal, and Matthew E. Taylor. 2019. On Hard Exploration for Reinforcement Learning: A Case Study in Pommerman. In *Proceedings of the 15th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press, 24–30.

[10] Bilal Kartal, Pablo Hernandez-Leal, Chao Gao, and Matthew E. Taylor. 2019. Safer Deep RL with Shallow MCTS: A Case Study in Pommerman. (2019). arXiv:1904.05759 [cs.LG]

[11] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *Proceedings of the 7th International Conference on Learning Representations*.

[12] Carol Luckhart and Keki B. Irani. 1986. An Algorithmic Solution of N-Person Games. In *Proceedings of the 5th National Conference on Artificial Intelligence*. Morgan Kaufmann, 158–162.

[13] Joseph Antonius Maria Nijssen. 2013. *Monte-Carlo Tree Search for Multi-Player Games*. Ph.D. Dissertation.

[14] J. (Pim) A. M. Nijssen and Mark H. M. Winands. 2011. Playout Search for Monte-Carlo Tree Search in Multi-player Games. In *Proceedings of the 13th International Conference on Advances in Computer Games (Lecture Notes in Computer Science, Vol. 7168)*. Springer, 72–83.

[15] Takayuki Osogami and Toshihiro Takahashi. 2019. Real-time tree search with pessimistic scenarios: Winning the NeurIPS 2018 Pommerman Competition. In *Proceedings of the 11th Asian Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 101)*. PMLR, 583–598.

[16] Sherjil Ozair, Yazhe Li, Ali Razavi, Ioannis Antonoglou, Aaron Van Den Oord, and Oriol Vinyals. 2021. Vector Quantized Models for Planning. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 8302–8313.

[17] Peng Peng, Liang Pang, Yufeng Yuan, and Chao Gao. 2018. Continual Match Based Training in Pommerman: Technical Report. (2018). arXiv:1812.07297 [cs.AI]

[18] Nick Petosa and Tucker Balch. 2019. Multiplayer AlphaZero. (2019). arXiv:1910.13012 [cs.AI]

[19] Cinjon Resnick, Wes Eldridge, David Ha, Denny Britz, Jakob N. Foerster, Julian Togelius, Kyunghyun Cho, and Joan Bruna. 2018. Pommerman: A Multi-Agent Playground. In *Joint Proceedings of the Workshops co-located with the 14th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (CEUR Workshop Proceedings, Vol. 2282)*. CEUR-WS.org/Vol-2282/MARLO_104.pdf

[20] Cinjon Resnick, Chao Gao, Görög Márton, Takayuki Osogami, Liang Pang, and Toshihiro Takahashi. 2020. Pommerman & NeurIPS 2018. In *The NeurIPS '18 Competition*, Sergio Escalera and Ralf Herbrich (Eds.). Springer International Publishing, Cham, 11–36.

[21] Cinjon Resnick, Roberta Raileanu, Sanyam Kapoor, Alexander Peysakhovich, Kyunghyun Cho, and Joan Bruna. 2018. Backplay: "Man muss immer umkehren". (2018). arXiv:1807.06919 [cs.LG]

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

[23] Maarten P. D. Schadd and Mark H. M. Winands. 2011. Best Reply Search for Multiplayer Games. *IEEE Transactions on Computational Intelligence and AI in Games* 3, 1 (2011), 57–66.

[24] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2020. Mastering atari, go, chess and shogi by planning with a learned model. *Nature* 588, 7839 (2020), 604–609.

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

[26] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354–359.

[27] Nathan R. Sturtevant and Richard E. Korf. 2000. On Pruning Techniques for Multi-Player Games. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*. AAAI Press / The MIT Press, 201–207.

[28] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. 2023. Monte Carlo Tree Search: a review of recent modifications and applications. *Artificial Intelligence Review* 56 (2023), 2497–2562.

[29] István Szita, Guillaume Chaslot, and Pieter Spronck. 2009. Monte-Carlo Tree Search in Settlers of Catan. In *Advances in Computer Games*. Springer, 21–32.

[30] Jannis Weil and Adrian Alic. 2021. pomcpp2: Pommerman Environment in C++. https://github.com/jw3il/pomcpp2.

[31] David J Wu. 2019. Accelerating Self-Play Learning in Go. (2019). arXiv:1902.10565 [cs.LG]

[32] H. Yang, S. Li, X. Xu, X. Liu, Z. Meng, and Y. Zhang. 2021. Efficient Searching With MCTS and Imitation Learning: A Case Study in Pommerman. *IEEE Access* 9 (2021), 48851–48859.