JMLR: Workshop and Conference Proceedings 101:583–598, 2019 ACML 2019

Real-time tree search with pessimistic scenarios: Winning the NeurIPS 2018 Pommerman Competition

Takayuki Osogami OSOGAMI@JP.IBM.COM
IBM Research - Tokyo

Toshihiro Takahashi E30137@JP.IBM.COM
IBM Research - Tokyo

Editors: Wee Sun Lee and Taiji Suzuki

Abstract

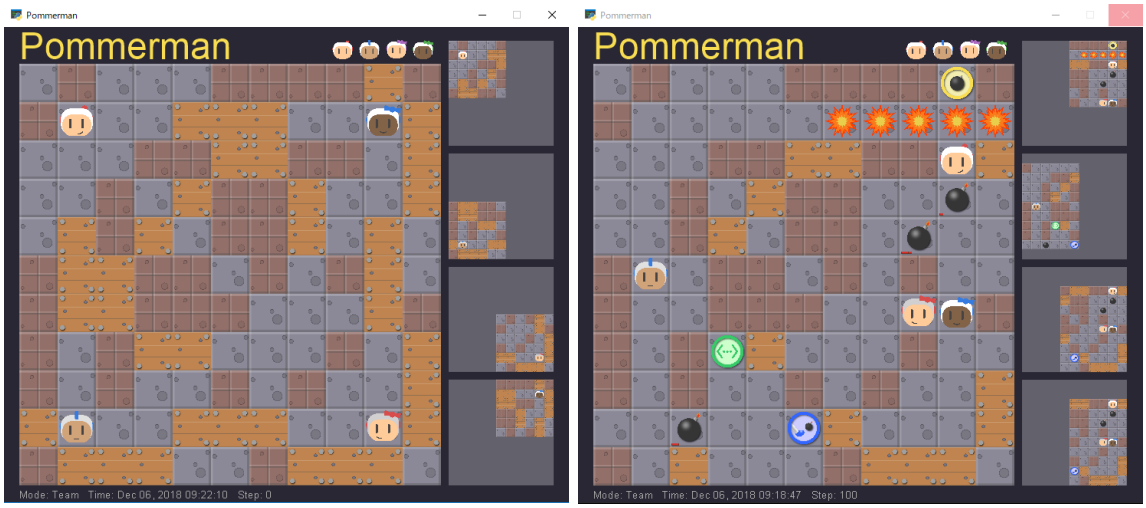
Autonomous agents need to make decisions in a sequential manner, under partially observable environment, and in consideration of how other agents behave. In critical situations, such decisions need to be made in real time for example to avoid collisions and recover to safe conditions. We propose a technique of tree search where a deterministic and pessimistic scenario is used after a specified depth. Because there is no branching with the deterministic scenario, the proposed technique allows us to take into account the events that can occur far ahead in the future. The effectiveness of the proposed technique is demonstrated in Pommerman, a multi-agent environment used in a NeurIPS 2018 competition, where the agents that implement the proposed technique have won the first and third places.

Keywords: Pommerman, Sequential decision making, Real-time, Tree search, Multi-agent, Partial observability.

1. Introduction

Autonomous agents, such as self-driving cars and drones, need to make decisions in real time (under tight time constraints), which is particularly important but difficult in critical situations for example to avoid collisions. Such decisions often need to be made in a sequential manner to achieve the eventual goal (*e.g.*, avoiding collisions and recovering to safe conditions), under partially observable environment, and by taking into account how other agents behave. Towards this far-reaching goal of realizing such autonomous agents, we propose practical techniques of sequential decision making in real time and demonstrate their effectiveness in Pommerman, a multi-agent environment that has been used in one of the competitions held at the Thirty-second Conference on Neural Information Processing Systems (NeurIPS 2018) on Dec. 8, 2018 (Resnick et al., 2018a). The techniques that we propose in this paper have been used in the Pommerman agents (HakozakiJunctions and dypm-final) who have won the first and third places in the competition.

In Pommerman, a team of two agents competes against another team of two agents on a board of 11×11 grids (see Figure 1 (a) for an initial configuration of the board). Each agent can observe only a limited area of the board, and the agents cannot communicate with each other. The goal of a team is to knock down all of the opponents. Towards this goal, the agents place bombs to destroy wooden walls and collect power-up items that might appear from those wooden walls, while avoiding flames and attacking opponents. See Figure 1 (b)



(a) Initial board (b) Board after 100 steps

Figure 1: An initial board (a) and a board after 100 steps in Pommerman. The four small windows on the right most column respectively denote the areas that the four agents can observe.

for an example of the board in the middle of the game. See Resnick et al. (2018a) and the GitHub repository[1] for details of Pommerman.

Although Pommerman has been developed recently (initial GitHub commit was Dec. 25, 2017), it has been gaining much attention as a benchmark of multi-agent study in the field of planning, game theory, and reinforcement learning (Kapoor, 2018; Hernandez-Leal et al., 2018). Prior to the one at NeurIPS 2018, the first competition was held on Jun. 3, 2018. The winning agent sets an intermediate goal with heuristics and performs depth-limited tree search to achieve that intermediate goal (Zhou et al., 2018). Resnick et al. (2018b) propose a technique of imitation learning with curriculum and show that, by using the behavioral data of the winning agent, it can train an agent as strong as the winning agent.

While the community has made progress in developing strong agents for Pommerman, the level of those agents is not yet comparable to what has been achieved for backgammon (Tesauro, 1994), Chess (Campbell et al.), Atari video games (Mnih et al., 2015), Poker (Bowling et al., 2015; Brown and Sandholm, 2019), and Go (Silver et al., 2017). Pommerman has its own difficulty that prohibits effective applications of existing approaches that have seen success in other games. For example, although deep reinforcement learning (Mnih et al., 2015) and planning methods (Lipovetzky et al., 2015) have seen success on Atari video games, they rely on the simulators of the video games at the phase of either learning or planning. These methods cannot be directly applied to (or less effective in) Pommerman, where opponents are not known in advance and cannot be simulated.

What makes Pommerman difficult is the constraint on real-time decision making (*i.e.*, an agent needs to choose an action in 100 milliseconds). This tight time constraint significantly

1. https://github.com/MultiAgentLearning/playground

limits the applicability of Monte Carlo Tree Search, which would otherwise be a reasonable approach to Pommerman (Matiisen, 2018). In Pommerman, the branching factor at each step can be as large as $6^4 = 1,296$, because four agents take actions simultaneously in each step, and there are six possible actions for each agent. The agents should plan ahead and choose actions by taking into account the explosion of bombs, whose lifetime is 10 steps. Tree search with insufficient depth (less than 10) would ignore the explosion of bombs, which in turn would make the agents easily caught up in flames. Tree search with sufficient depth (at least 10) is practically infeasible with the large branching factor. Other difficulties of Pommerman include the following. Reward is only given at the end of an episode, which can be as long as 800 steps. The agents can observe only a limited part of the board, and some of the key information cannot be directly observed. The agent needs to coordinate with its teammate without explicit communication.

Here, we propose a practical approach to real-time tree search that allows us to take into account critical events that can occur far ahead in the future. In our approach, tree search after a specified depth is performed under the assumption of a deterministic and pessimistic scenario (*i.e.*, sequence of states). Because the scenario is deterministic, there is no branching after the specified depth, which allows us to perform the tree search with sufficient depth to take into account the critical events in the distant future. This deterministic scenario is designed to be pessimistic by allowing multiple unfavorable events can happen (*e.g.*, by letting opponents take multiple actions) simultaneously in a nondeterministic manner. Hence, our pessimistic scenarios are unrealistic in general. Our key idea is that an unrealistic scenario can capture critical events in the future better than a small number of realistic scenarios that can be sampled and explored under the tight time constraint. We adjust the level of pessimism via self-play to achieve the best overall performance.

Our approach is proposed particularly for Pommerman but can be generally applicable to other domains that require real-time sequential decision making under tight time constraints. We demonstrate the flexibility of the proposed approach by instantiating it as two variants of Pommerman agents, who need to deal with the complex environment that involves multiple agents and partial observability. The effectiveness of the proposed approach is shown with Pommerman. The new approach of real-time tree search with deterministic and pessimistic scenarios and its application to Pommerman constitute the contributions of this paper.

2. Related Work

There has been a significant amount of work on the techniques of tree search for real-time (strategy) games. As we will discuss it in the following, however, the focus of the prior work is on the techniques for reducing the search space or guiding the search towards the most relevant subspace. The novelty in our approach is in synthesizing the deterministic and pessimistic scenarios.

The prior work has investigated various techniques to make Monte Carlo Tree Search (MCTS) applicable to real-time games such as Ms. Pac-Man (Pepels et al., 2014; Ikehata and Ito, 2011), StarCraft (Uriarte and Ontañón, 2016), Wargus (Balla and Fern, 2009), Physical Traveling Salesman Problem (Powley et al., 2012), Quantified Constraint Satisfaction Problem (Baba et al., 2011), and μRTS (Barriga et al., 2018; Mariño et al., 2019).

An example of a recent work in this line is Mariño et al. (2019), who study a technique of action abstraction and apply it to MCTS among others to reduce the search space. Barriga et al. (2018) study a technique of using non-deterministic rules to reduce the branching in MCTS. In all of such prior work, MCTS is performed only with realistic or legal moves.

Guo et al. (2014) study the approach of using MCTS to generate training data for learning a deep neural network that approximates a policy or a value function, and the effectiveness of the proposed approach is demonstrated in Atari video games. This approach is motivated by the observation that tree search (planning-based approaches) can perform far better than model-free approaches if it were not for the tight time constraint.

Real-time tree search has also been studied for deterministic settings. Here, the search tree is expanded on the basis of heuristic values as long as time permits. Similar to real-time MCTS, key questions are where to expand and what actions to take given the search tree investigated. For example, Mitchell et al. (2019) propose a risk-sensitive approach to these questions.

Our approach of synthesizing a pessimistic scenario is also related to the null-move heuristic, which has been studied particularly for Chess (Goetsch and Campbell, 1990), in that it considers a scenario with illegal moves. The null-move heuristic assumes that a player skips a move, which is illegal in chess, to estimate a lower bound of the value of the best move. The lower bound is then used to prune the search space. In contrast, our approach can assume that a player takes multiple moves in the pessimistic scenario.

A pessimistic scenario is also similar to delete relaxation or relaxed plan heuristics (Hoffmann and Nebel, 2001) in classical planning. Delete relaxation is similar to a pessimistic scenario in the sense that it does not "delete" an opponent from a position. A difference is that a pessimistic scenario allows an opponent to take multiple actions simultaneously.

Our approach is also similar to "variable resolution" (Martínez et al., 2016) in that exact search is limited to a certain depth. After that depth, the two approaches differ. In particular, a pessimistic scenario cannot be obtained by "removing some information from the planning task," as is done in Martínez et al. (2016).

3. Real-time tree search for Pommerman

In Pommerman, the dynamics of the environment is known, and much of the uncertainties resulting from partial observability can be resolved with careful analysis of historical observations. MCTS would thus be a competitive approach if it were not for the tight time constraint (Matiisen, 2018). For example, consider a situation where an agent can survive only by following a particular route. Tree search is particularly suitable for finding such a route, while model-free approaches of learning policies or value functions, if not impossible, would require large scale functional approximators (*e.g.*, deep neural networks) and a large amount of data for training to be able to follow that route. The applicability of MCTS or tree search in general is however significantly limited in Pommerman due to the tight time constraint and the large branching factor.

One approach of tree search is to push the depth as far as possible, and this is the approach taken by the gorogm_eisenach (`eisenach`) agents, who won the second place in the NeurIPS 2018 Pommerman competition. The `eisenach` agent was implemented in C++

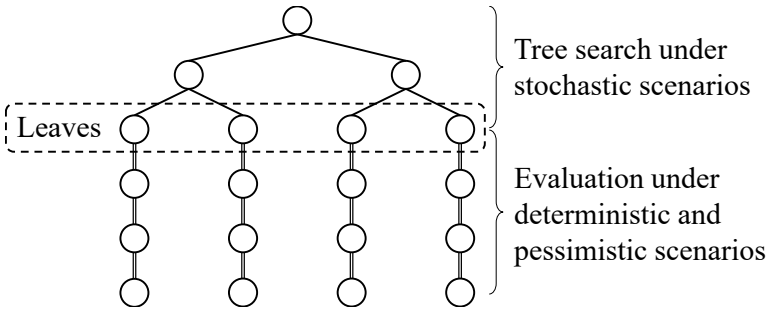


Figure 2: Tree search with deterministic and pessimistic scenarios.

with various engineering tricks to achieve the average depth of 2 in the tree search (Resnick et al., 2019).

3.1. Tree search with pessimistic scenarios

In our approach, the tree search after a specified depth is performed under a deterministic and pessimistic scenario, which will also be simply referred to as a deterministic scenario or a pessimistic scenario, depending on what feature of the scenario is more relevant in the context. Figure 2 shows an example. Here, the tree search is performed in a standard manner until the depth of 2. In this example, the branching factor is 2, and there are 4 nodes at the depth of 2. From each of these 4 nodes, "tree search" is continued until it reaches the depth of 5 by assuming a deterministic scenario. Because the scenario is deterministic, there are no branches after the depth of 2.

One may also interpret our approach as evaluating the 4 leaves (at the depth of 2) on the basis of the deterministic scenario from the depth of 2 to the depth of 5, and our following discussion will be based on this interpretation. Our approach keeps the size of the search tree small, because there are branches only until a limited depth L. At the same time, our approach can take into account the events that can occur far ahead in the future, because the leaves (nodes at depth L) can be evaluated with a deterministic scenario that can be much longer than what would be possible with branches. What differs from the rollout in MCTS is that we let the deterministic scenario be pessimistic, as we will discuss in Section 3.2.

More specifically, our approach performs standard tree search (*e.g.*, MCTS, exhaustive tree search, etc.) but on the search tree of a limited depth L, evaluating the leaves (nodes at the depth L) with pessimistic scenarios, and select the best action for the root node (*e.g.*, as is selected with minimax tree search or with multi-player tree search such as $\max^n$ tree search (Luckhardt and Irani, 1986), paranoid search algorithm (Sturtevant and Korf, 2000), and best-reply search (Schadd and Winands, 2011)). Namely, *our approach can use any tree search algorithms for multi-player games but limit the depth of the search tree at a given L and evaluate the leaves with pessimistic scenarios.* One can choose the value of L in consideration of real-time constraints.

Algorithm 2 shows the general framework of our real-time tree search with pessimistic scenarios. Notice that, except at the depth of L, `PessimisticTreeSearch` is nothing but the standard tree search for games, and this tree search may also be done non-exhaustively

```
PessimisticTreeSearch(node)
Input: node is the root node where tree search starts; L is the depth of tree search under
       stochastic scenarios
Output: The best action at node
if depth of node = L then
    /* Evaluate node under a pessimistic scenario                              */
    board ← GetState(node)                          // Get the game state at node
    board_sequence ← PessimisticSimulation(board)
    score(node) ← Evaluate(board_sequence)
else if all children of node have been evaluated then
    if depth of node = 0 then
        Find the bast action at node based on the score of its children
        Return: the best action at node
    else
        /* Compute the score of node from the scores of its children          */
        score(node) ← GetScore(node)
    end
end
node ← FindUnevaluatedNode()          // e.g., via depth/breadth first search
PessimisticTreeSearch(node)                          // Recursively run from node
```

Algorithm 1: The general framework of the real-time tree search with pessimistic scenarios

similar to MCTS. For a node at depth L, `PessimisticTreeSearch` evaluates the node under a pessimistic scenario, which relies on two subroutines: `PessimisticSimulation` and `Evaluate`. Given a state of the game, `PessimisticSimulation` generates a sequence of states from that state in a pessimistic manner. Although Pommerman is an imperfect information game and an information set can be associated with a node, we will associate a single state to the node by resolving uncertainties, as we will discuss in Section 3.2. Then the forward model of Pommerman can be used to simulate the transitions of states. Specific procedures of `PessimisticSimulation` will be described in Section 3.2. `Evaluate` gives the score of the leaf on the basis of the sequence of the states given by `PessimisticSimulation`. Specific procedures of `Evaluate` will be described in Section 3.3.

More specifically, in the Pommerman agents that implement the proposed approach (*i.e.*, HakozakiJunctions (`hakozaki`) and dypm-final (`dypm`)), the depth of tree search is set as $L = 1$. The `hakozaki` agent considers all of the leaves at depth 1 but might need to choose an action before exhaustively searching all of the leaves due to timeout. On the other hand, the `dypm` agent considers six leaves at depth $L = 1$ by taking into account only its own actions, and the effect of the the actions by the other agents are taken into account in evaluation with the deterministic scenario. In both of `hakozaki` and `dypm`, the leaves are evaluated with a deterministic scenario with the length of at least 10 to take into account the explosion of bombs, whose lifetime is 10. Recall that the `eisenach` agent can perform the tree search only at the average depth of $L = 2$. Hence, if we performed the standard tree search for 2 steps, there would be no computational budget left for the evaluation with deterministic scenarios.

Input: `leaf` is the leaf to evaluate; `length` is the length of a pessimistic scenario; `pessimism_level` is the level of pessimism;
Output: The sequence of states from the leaf under a pessimistic scenario

```
boards[0] ← GetState(leaf)                    // Get the state of the game at leaf
/* Sample a sequence of states from the given state                          */
for ℓ = 1, ..., length do
    boards[ℓ] ← ForwardModel(boards[ℓ − 1])        // One of next states of game
end
/* Make the sequence of states pessimistic                                  */
for ℓ = 1, ..., pessimism_level do
    for each object in boards[ℓ − 1] do
        positions ← Find the next positions of object other than the one in boards[ℓ]
        for each position in positions do
            Copy and place object at position in boards[ℓ]
        end
    end
end
```

Return: `boards`

Algorithm 2: An example of `PessimisticSimulation`, which generates a pessimistic scenario for the case where the state of the game can be represented by the positions of objects.

3.2. Generating pessimistic scenarios

From each of the leaves in the search tree, we generate a deterministic scenario. A key idea in our approach is to make this deterministic scenario be pessimistic, which we will discuss in this section with specific instantiation as Pommerman agents.

A pessimistic scenario can be generated in a systematic manner as follows. We assume that the state of the environment can be represented by the positions of objects. In Pommerman, these objects are agents, bombs, flames, power-up items, and walls. Some of those objects change their positions randomly or by depending on the actions of the agents, which forces the search tree to have branches. If one can tell the worst sequence of the positions of an object among all of the possibilities, one can place and move that object accordingly in the pessimistic scenario. It is often the case, however, that the worst positions are unknown. Instead, *we generate a pessimistic scenario by allowing the objects to be located at multiple positions* even if that is unrealistic or illegal. In Pommerman, this typically corresponds to assuming that an opponent takes multiple actions simultaneously in a nondeterministic manner, which means that the opponent is copied into multiple positions in the next step. Notice that such generated pessimistic scenarios can be more adversarial than assuming the worst possible scenarios, because an object cannot actually be at multiple positions.

Algorithm 2 shows an example of how a pessimistic scenario is generated from the leaf (node at depth L) of the search tree. This `PessimisticSimulation` samples a sequence of states from the leaf node, similar to rollout in MCTS, but the sequence of states is then made pessimistic by allowing an object to be placed at multiple positions.

Algorithm 2 involves two hyper-parameters: `length` and `pessimism_level`. Here, `length` denotes the length of the pessimistic scenario used to evaluate the given leaf. One

can choose the value of `length` in consideration of real-time constraints. On the other hand, `pessimism_level` denotes the level of pessimism and controls how pessimistic the pessimistic scenario is. The value of `pessimism_level` can be optimized via self-play in a way that the overall performance is maximized, which will be further discussed in the following.

What is essential is that a pessimistic scenario is deterministic to avoid the computational complexity resulting from branching in the search tree. Our guideline is to make this deterministic scenario rather pessimistic, because good actions are often the ones that perform well under pessimistic scenarios particularly in cases where safety is a primary concern. In Pommerman, an agent dies if it cannot escape from flames, and our team loses if both of our agents die. It is thus of critical importance to ensure that our agents can survive, while attacking opponents or collecting power-up items.

In Pommerman, a deterministic scenario is represented by a sequence of boards, where each board is given by the state of the game at a certain point in time. Such a deterministic scenario can be generated by a forward model of Pommerman after resolving uncertainties. There are two sources of uncertainties: the future actions of agents and partial observability. These uncertainties can be resolved in arbitrary ways, but our guideline is to resolve them in rather pessimistic manners and to optimize the level of pessimism by tuning hyperparameters via self-play. A caveat is that the purpose of a pessimistic scenario is not to find a proper lower bound on the value of a leaf but to find a good action as a result of evaluating that leaf with the pessimistic scenario. Therefore, a pessimistic scenario can be more adversarial or less adversarial than the worst scenario. The self-play allows us to optimize the level of pessimism.

More specifically, both of `hakozaki` and `dypm` agents generate a sequence of boards by letting the other agents move to multiple positions simultaneously. They then record the time when each position is first occupied by an agent. There are differences between `hakozaki` and `dypm` in exactly what information is recorded in the sequence of boards. In `hakozaki`, each position in the t-th board has the information about when that position was occupied, if the position has been occupied by the t-th step. In `dypm`, each position in the t-th board has the information about whether that position has been occupied by the t-th step. Also, `dypm` assumes that the other agents take actions only for a predetermined number of steps (a hyperparameter tuned via self-play), while `hakozaki` does not have this limit.

Note that the sequence of such boards is in general illegal or unrealistic. There may be multiple copies of an agent in the board (`dypm`), and an agent that might occupy a position may be replaced by the integer value representing when that position can be occupied (`hakozaki`). Also, some of the uncertainties are resolved in a way that is not necessarily pessimistic. For example, we ignore the possibility that an agent might kick bombs in the sequence of boards. However, the purpose of the sequence of boards is not to compute a proper lower bound of the value but to quickly estimate the relative values of the leaves in the search tree in a way that it gives the overall best performance. Note that the part of tree search (with branches) can take into account all of the details, unlike the part of evaluation with a deterministic scenario. For example, if the action of an agent within the tree search is to kick a bomb (by moving to the bomb), the movement of the kicked bomb is taken into account in the remaining tree search as well as in the deterministic scenario.

Our approach of tree search with a pessimistic scenario allows us to take into account all of the details in the near future (via tree search) as well as possibly critical events in the distant future (via a pessimistic scenario).

3.3. Evaluation with pessimistic scenarios

This sequence of boards is then used to estimate the value of the initial board in the sequence (*i.e.*, a leaf). If we obtain reward sufficiently frequently, we could use Monte Carlo return (cumulative reward obtained during the sequence of boards) as an estimate of the value. However, in Pommerman, reward is obtained only at the end of a game, and the Monte Carlo return has very high variance. We thus design the value in a way that choosing actions that give high value tends to eventually achieving the goal. The goal of a Pommerman agent is to knock down all of the opponents, while the agent or its teammate is surviving. The value should thus reflect some notion of the survivability of the agent itself, its teammate, and its opponents. Roughly speaking, high value should imply high survivability of the agent itself and its teammate, low survivability of the opponents, or both.

In Pommerman, the survivability of an agent can be captured by the number of positions that the agent can stay safely in the sequence of boards. More specifically, given a deterministic scenario, `dypm` counts the number of the time-position pairs from which an agent can survive at least until the end of the scenario. This number is used as the survivability of the agent. Namely, the survivability of the agent is computed by first searching the reachable time-position pairs in the sequence of boards and then pruning those pairs from which one cannot survive until the end of the sequence. On the other hand, `hakozaki` finds the positions that an agent can reach at the end of the sequence of boards, and compute the survivability on the basis of the integer values that represent when the positions might be occupied by other agents. Intuitively, an agent is considered to have high survivability if there are many positions that the agent can reach without contacting the other agents.

Note that an agent i computes the survivability $S(j, s)$ for each leaf (state) s and for each agent j who is visible from i, including i itself. The survivability of an agent $j \neq i$ is computed on the basis of the sequence of boards that is pessimistic to j (*i.e.*, the agents except j move to multiple positions simultaneously)[2].

Now, one can choose the best action on the basis of these survivabilities. Roughly speaking, our agent chooses the action that maximizes the product the survivabilities of the agent itself[3] and its teammate divided by the product of the survivabilities of the opponents. Because the survivability is defined for each leaf, which corresponds to a combination of the actions of all agents, one needs to marginalize out the actions of the other agents to define the survivability of an agent with a particular action. The survivability of an agent when that agent takes a particular action can be defined to be the minimum survivability of that agent given that the agent takes that action (*i.e.*, worst case). The survivability of a teammate can also be defined as the minimum survivability. On the other hand, we find that the survivability of an opponent with that action should be defined as the average

2. In `dypm`, to save computational cost, a single sequence of boards with no move of agents is used to compute the survivabilities of agents $j \neq i$, and those survivabilities are normalized by dividing them by the corresponding survivabilities when the agent i does not exist.
3. To be more aggressive, the `dypm` agent clips its own survivability S at a threshold S_{th} when S exceeds S_{th}, where S_{th} is tuned via self-play.

survivability rather than minimum or maximum. These are how the survivabilities with a particular action are defined in `hakozaki`. On the other hand, each leaf in the search tree of a `dypm` agent corresponds to each action of that agent, and there is no need for marginalization. A caveat is that the action by the `dypm` agent might be blocked by other agent, resulting in no move. The survivability with such an action is thus averaged with the survivability of no move.

In this section, we have discussed how our agents choose actions in most critical situations of Pommerman, where the agents interact with other agents. In those situations, the goal of an agent is to reduce the survivabilities of the opponents, while keeping the survivabilities of the agent itself or its teammate sufficiently high. In other situations, the agent can ignore the behavior of other agents and seek to attain intermediate objectives. Examples of such objectives include (i) breaking wooden walls to make passages or to find power-up items, (ii) collecting power-up items, and (iii) moving towards the areas that cannot been observed to obtain new information. Although there is a question of what objective to pursue at each step, it is relatively easy to choose actions once an objective is given, because we do not need to take into account the actions of the other agents. Our agent heuristically sets an objective and chooses actions by the use of a standard search technique until the agent meets and starts interacting with other agents.

4. Experiments

We conduct two sets of experiments to investigate the overall performance of the proposed approach and the effectiveness of our key idea (*i.e.*, the use of pessimistic scenarios in tree search). Although the agents that implement our proposed approach have won the first and third places in the NeurIPS 2018 Pommerman competition, the number of matches in the competition is too limited to draw conclusions. In the first set of experiments, we intensively evaluate the performance among the top five teams, from the competition, that implement state-of-the-art approaches, including ours. In the second set of experiments, we study the effectiveness of pessimism by changing one of the key hyperparameters of `dypm` that controls the level of pessimism in the proposed approach.

4.1. Performance against state-of-the-art agents

The competition was run according to a double elimination style with two brackets, where the team that won two games before the other moves on to the next round. A tie was replayed for the first time, but another tie was resolved by matches on another version of Pommerman, where walls can collapse. Namely, tie was not the same as "no game" in the competition, and the results in this section needs to be interpreted accordingly. See Resnick et al. (2019) for more details about the settings of the competition.

The top five teams were `hakozaki`, `eisenach`, `dypm`, `navocado`, and `skynet`. The top three teams are based on tree search, as we have discussed in Section 3. The other two are based on reinforcement learning. More specifically, `navocado` is based on advantage-actor-critic (Peng et al., 2018), and `skynet` is based on proximal policy optimization.

Table 1 shows the results of the direct matches among the top five teams in the competition. For example, `eisenach` defeated `dypm` four times, `dypm` defeated `eisenach` once, and there were no ties between these two teams. Although the winners of the competition were

determined according to the rule of the competition, the number of matches was clearly limited to statistically determine which teams are stronger than others. In particular, there were pairs of teams that had no direct matches in the competition.

	hakozaki⋆	eisenach	dypm⋆	navocado	skynet
hakozaki⋆	-	4/2/1	-	2/0/2	2/0/0
eisenach	2/4/1	-	4/1/0	-	-
dypm⋆	-	1/4/0	-	2/1/0	-
navocado	0/2/2	-	1/2/0	-	2/1/5
skynet	0/2/0	-	-	1/2/5	-

Table 1: A summary of the matches among the top five teams in the competition. Each entry shows the number of "wins / losses / ties" for a row agent against a column agent. The ⋆ marks indicate the teams that implement our approach.

The purpose of the following experiments is to complement the competition by running a greater number of matches between the top five teams. We use the Docker images[4] of the agents that have been used in the competition. We run our experiments on a Ubuntu 18.04 machine having eight Intel Core i7-6700K CPUs running at 4.00 GHz and 64 GB random access memory. Note that these computational resources are different from what has been used at the competition (exact computational resources used at the competition are not known). Therefore, the results of our experiments should not be considered as a refinement but rather as complementary to the results from the competition.

Table 2 summarizes the results of the 1,000 matches that we have run between each pair of the teams. Because there are two essentially different configurations for the initial positions of the two teams, half of the matches are initialized with one configuration, and the other with the other configuration. Each team is also matched against itself for 500 matches, and the results for both sides are counted in the table (if a match is tied, two ties are counted). In total, we run 22,500 matches, which take approximately two weeks in our environment.

In our experiments, the top three teams (`hakozaki`, `eisenach`, `dypm`) have completely dominated the other two (`navocado`, `skynet`), recording 5,227 wins (87.1 %), 162 losses (2.7 %), and 611 ties (10.2 %). In particular, `hakozaki` and `dypm`, who implement our proposed approach, have lost against `navocado` or `skynet` only in 22 matches (0.4 %). While the top three teams are based on tree search, the other two are based on model-free reinforcement learning. This indicates the advantages of tree search in Pommerman, where precise planning in the following several steps is critically important to survive from the explosion of bombs. The results in our experiments are not necessarily consistent with those in the competition (Table 1), however. In particular, `dypm` has lost once against `navocado` in the competition. This may be because the `dypm` agents have occasionally experienced timeouts (*i.e.*, the agent does not respond in 100 milliseconds, which is treated

4. The Docker images are available as multiagentlearning/{hakozakijunctions, eisenach, dypm.1, dypm.2, navocado, skynet955}. Note that the `dypm` team uses two agents, dypm.1 and dypm.2, that differ only in the values of their hyperparameters. In the other teams, the two agents are identical.

	hakozaki*	eisenach	dypm*	navocado	skynet	TOTAL
hakozaki*	112/112/776	490/139/371	279/221/500	694/ 3/303	845/ 7/148	2420/482/2098
eisenach	139/490/371	338/338/324	403/492/105	866/ 85/ 49	918/ 55/ 27	2664/1460/876
dypm*	221/279/500	492/403/105	107/107/786	935/ 12/ 53	969/ 0/ 31	2724/801/1475
navocado	3/694/303	85/866/ 49	12/935/ 53	95/ 95/810	198/ 50/752	393/2640/1967
skynet955	7/845/148	55/918/ 27	0/969/ 31	50/198/752	57/ 57/886	169/2987/1844

Table 2: A summary of the 1,000 matches between each pair among the top five teams. Each entry shows the number of "wins / losses / ties" for a row agent against a column agent. The rightmost column shows the total number of "wins / losses / ties" for row agents. The $\star$ mark indicates the team that implements the proposed approach.

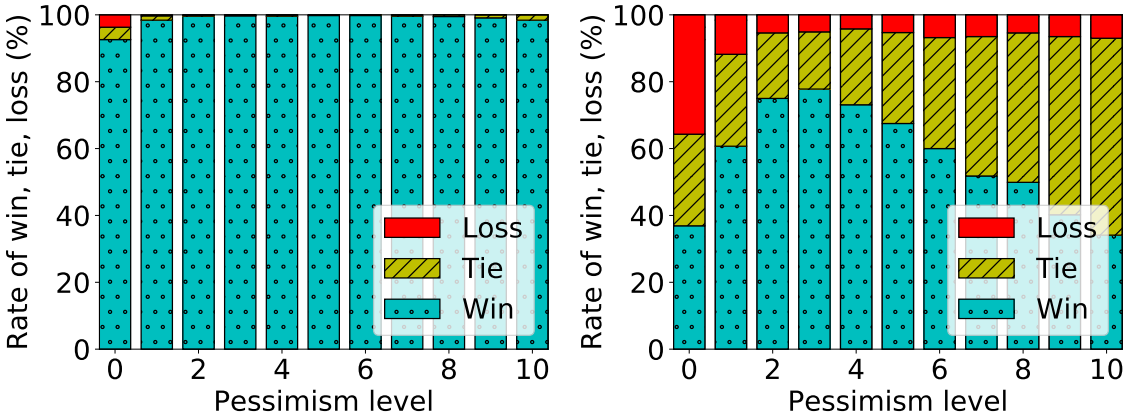
as a "stop" action) in the competition, because `dypm` does not implement any mechanisms for measuring the elapsed time. Also, the computational resources in our experiments might be less favorable to the learning agents (`navocado` and `skynet`) than what has been used in the competition.

Among the top three teams, `hakozaki` has consistently dominated the others, although the relative advantages between `hakozaki` and `dypm` are relatively small. Also, `dypm` has slightly outperformed `eisenach`. Overall, `hakozaki` and `dypm`, who implement the proposed approach, have recorded 982 wins (49.1 %), 542 losses (27.1 %), and 476 ties (23.8 %) against `eisenach`. Note that these top three teams implement their agents and forward models in different programming languages: `hakozaki` in Java™, `eisenach` in C++, and `dypm` in Python. Also, the `dypm` agent runs on a single thread, while `hakozaki` and `eisenach` use multiple threads. Overall, our experimental results suggest the superiority of the proposed approach in real-time tree search for sequential decision making with limited computational resources.

4.2. Effectiveness of pessimism

We next study the effectiveness of the pessimism in the deterministic scenario used in our proposed approach. Recall that a `dypm` agent generates a deterministic scenario by assuming that the other agents take multiple actions simultaneously in a nondeterministic manner but only for a limited number of steps. Here, we refer to this limited number of steps as the pessimism level and study how the performance of `dypm` depends on the pessimism level. Specifically, for each pessimism level, we run 1,000 matches against a baseline and record the number of wins, losses, and times. The baseline is either a team of default agents (`SimpleAgent`) or a team of the `dypm` agents whose pessimism level is set 0.

Figure 3 shows the number of wins, losses, and ties of `dypm` against each baseline, where the pessimism level in `dypm` is varied as indicated along the horizontal axis. For example, `dypm` with the pessimism level 3 has had 997 wins, 1 loss, and 2 ties against `SimpleAgent` (Figure 3 (a)), and this pessimism level is the one that has actually been set in `dypm` in the competition. The winning rate of `dypm` against `SimpleAgent` increases from 92.6 % to



(a) vs. `SimpleAgent` (b) vs. no pessimism

Figure 3: The performance of `dypm` with varying pessimism levels. The rate of wins, losses, and ties of `dypm` against a baseline is shown for each pessimism level (from 0 to 10) as indicated along the horizontal axis. The baseline is `SimpleAgent` in (a) and the `dypm` with no pessimism (level 0) in (b).

99.7 % by increasing the pessimism level from 0 to 3. Further increasing the pessimism level can reduce the number of losses but increases the number of ties, but these changes are insignificant with the very high winning rate.

To clarify these changes, Figure 3 (b) shows the results against a stronger baseline, which is `dypm` with pessimism level 0. Now, the winning rate of `dypm` against the baseline increases from 36.9 % to 77.8 % by increasing the pessimism level from 0 to 3. Further increasing the pessimism level from 3 to 4 can reduce the rate of losses from 5.1 % to 4.2 % but increases the rate of ties from 17.1 % to 22.7 %.

Overall, we find that pessimism in deterministic scenarios can significantly improve the overall performance of Pommerman agents. Also, the performance is sensitive to the particular level of pessimism, and an appropriate level of pessimism may be determined via self-play. Note, however, that Pommerman is an extensive-form game with imperfect information, and the optimal strategy in general is to probabilistically mix multiple levels of pessimism. Also, `dypm` has other hyperparameters, and the optimal level of pessimism depends on the values of the other hyperparameters.

5. Conclusion

We have proposed an approach of real-time tree search, where tree search is performed only with a limited depth, and the leaves are evaluated under a deterministic and pessimistic scenario. Because there is no branching with a deterministic scenario, our evaluation can take into account the events that can occur far ahead in the future. Also, evaluation with pessimistic scenarios can lead to selecting good actions, which are often the ones that perform well under the pessimistic scenarios particularly in cases where safety is a primary concern. We have assumed that the state can be represented by the positions

of objects and generated pessimistic scenarios by allowing the objects to be located at multiple positions even if that may be unrealistic or illegal. One could, however, apply the general idea of our real-time tree search with pessimistic scenarios to a broader range of domains by designing pessimistic scenarios suitable for particular domains. For example, for applications to autonomous robots, drones, or other agents in continuous space, a pessimistic scenario may be generated by assuming objects (e.g., other robots and drones) that increases the size over time.

Our experiments with Pommerman suggest that the performance of the proposed approach is sensitive to the particular level of pessimism, but it can be optimized via self-play. With the optimized level of pessimism, the proposed approach is shown to outperform other state-of-the-art approaches to real-time sequential decision making. An interesting direction of future work is to combine the proposed approach with model-free reinforcement learning, where the proposed approach is used to choose specific actions in each step to attain the intermediate objective that is selected by model-free reinforcement learning. Such integration of tree search and reinforcement learning is an active field of research (Silver et al., 2017; Anthony et al., 2017; Vodopivec and nd B. Šter, 2017; Jiang et al., 2018; Efroni et al., 2019; Anthony et al., 2019; Shah et al., 2019).

Acknowledgments

Takayuki Osogami was supported in part by JST CREST Grant Number JPMJCR1304, Japan.

References

T. Anthony, Z. Tian, and D. Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems 30*, pages 5360–5370. 2017.

T. Anthony, R. Nishihara, P. Moritz, T. Salimans, and J. Schulman. Policy gradient search: Online planning and expert iteration without search trees. *CoRR*, abs/1904.03646, 2019.

S. Baba, Y. Joe, A. Iwasaki, and M. Yokoo. Real-time solving of quantified CSPs based on Monte-Carlo game tree search. In *Proc. 22nd International Joint Conference on Artificial Intelligence*, pages 655–661, 2011.

R.-K. Balla and A. Fern. UCT for tactical assault planning in real-time strategy games. In *Proc. 21st International Joint Conference on Artifical intelligence*, pages 40–45, 2009.

N. A. Barriga, M. Stanescu, and M. Buro. Game tree search based on nondeterministic action scripts in real-time strategy games. *IEEE Trans. Games*, 10:69–77, 2018.

M. Bowling, N. Burch, M. Johanson, and O. Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347:145–149, 2015.

N. Brown and T. Sandholm. Superhuman AI for multiplayer poker. *Science*, 10.1126/science.aay2400, 2019.

M. Campbell, A. J. Hoane Jr, and F. Hsu. Deep Blue. *Artificial intelligence*, 134:57–83.

Y. Efroni, G. Dalal, B. Scherrer, and S. Mannor. How to combine tree-search methods in reinforcement learning. In *Proc. 33rd AAAI Conference on Artificial Intelligence*, 2019.

G. Goetsch and M. S. Campbell. Experiments with the null-move heuristic. In *Computers, Chess, and Cognition*, pages 159–168. Springer-Verlag, 1990.

X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. In *Advances in Neural Information Processing Systems 27*, pages 3338–3346. 2014.

P. Hernandez-Leal, B. Kartal, and M. E. Taylor. Is multiagent deep reinforcement learning the answer or the question? A brief survey. *CoRR*, abs/1810.05587, 2018.

J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

N. Ikehata and T. Ito. Monte-Carlo tree search in Ms. Pac-Man. In *Proc. 2011 IEEE Conference on Computational Intelligence and Games*, pages 39–46, 2011.

D. Jiang, E. Ekwedike, and H. Liu. Feedback-based tree search for reinforcement learning. In *Proc. 35th International Conference on Machine Learning*, pages 2284–2293, 2018.

S. Kapoor. Multi-agent reinforcement learning: A report on challenges and approaches. *CoRR*, abs/1807.09427, 2018.

N. Lipovetzky, M. Ramirez, and H. Geffner. Classical planning with simulators: Results on the Atari video games. In *Proc. 24th International Conference on Artificial Intelligence*, pages 1610–1616, 2015.

C. A. Luckhardt and K. B. Irani. An algorithmic solution of n-person games. In *Proc. the 5th National Conference on Artificial Intelligence*, pages 158–162, 1986.

J. Mariño, R. Moraes, C. Toledo, and L. Lelis. Evolving action abstractions for real-time planning in extensive-form games. In *Proc. 33rd AAAI Conference on Artificial Intelligence*, 2019.

M. Martínez, F. Fernández, and D. Borrajo. Planning and execution through variable resolution planning. *Robotics and Autonomous Systems*, 83:214–230, 2016.

T. Matiisen. Pommerman baselines. https://github.com/tambetm/pommerman-baselines, 2018.

A. Mitchell, W. Ruml, F. Spaniol, J. Hoffmann, and M. Petrik. Real-time planning as decision-making under uncertainty. In *Proc. 33rd AAAI Conference on Artificial Intelligence*, 2019.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C . Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

P. Peng, L. Pang, Y. Yuan, and C. Gao. Continual match based training in Pommerman: Technical report. *CoRR*, abs/1812.07297, 2018.

T. Pepels, M. H. M. Winands, and M. Lanctot. Real-time Monte Carlo tree search in Ms Pac-Man. *IEEE Trans. Computational Intelligence and AI in Games*, 6:245–257, 2014.

E. J. Powley, D. Whitehouse, and P. I. Cowling. Monte Carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem. In *Proc. 2012 IEEE Conference on Computational Intelligence and Games*, pages 234–241, 2012.

C. Resnick, W. Eldridge, D. Ha, D. Britz, J. Foerster, J. Togelius, K. Cho, and J. Bruna. Pommerman: A multi-agent playground. *CoRR*, abs/1809.07124, 2018a.

C. Resnick, R. Raileanu, S. Kapoor, A. Peysakhovich, K. Cho, and J. Bruna. Backplay: "Man muss immer umkehren". *CoRR*, abs/1807.06919, 2018b.

C. Resnick, C. Gao, G. Marton, T. Osogami, L. Pang, and T. Takahashi. Pommerman: A multiagent playground. In *The NeurIPS 2018 Competition*, The Springer Series on Challenges in Machine Learning. Springer, 2019.

M. P. D. Schadd and M. H. M. Winands. Best reply search for multiplayer games. *IEEE Trans. Computational Intelligence and AI in Games*, 3:57–66, 2011.

D. Shah, Q. Xie, and Z. Xu. On reinforcement learning using Monte Carlo tree search with supervised learning: Non-asymptotic analysis. *CoRR*, abs/1902.05213, 2019.

D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, pages 354–359, 2017.

N. R. Sturtevant and R. E. Korf. On pruning techniques for multi-player games. In *Proc. 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 201–207, 2000.

G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

A. Uriarte and S. Ontañón. Improving Monte Carlo tree search policies in StarCraft via probabilistic models learned from replay data. In *Proc. 12th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, pages 100–106, 2016.

T. Vodopivec and S. Samothrakis nd B. Šter. On Monte Carlo tree search and reinforcement learning. *Journal of Artificial Intelligence Research*, 60:881–936, 2017.

H. Zhou, Y. Gong, L. Mugrai, A. Khalifa, A. Nealen, and J. Togelius. A hybrid search agent in Pommerman. In *Proc. 13th International Conference on the Foundations of Digital Games*, Article No. 46 2018.