Boney, Rinu; Ilin, Alexander; Kannala, Juho; Seppanen, Jarno

Learning to Play Imperfect-Information Games by Imitating an Oracle Planner

Published in:
IEEE Transactions on Games

DOI:
10.1109/TG.2021.3067723

Published: 01/01/2022

Document Version
Publisher's PDF, also known as Version of record

Learning to Play Imperfect-Information Games by Imitating an Oracle Planner

Rinu Boney, Alexander Ilin, Juho Kannala, and Jarno Seppänen

***Abstract*—We consider learning to play multiplayer imperfect-information games with simultaneous moves and large state-action spaces. Previous attempts to tackle such challenging games have largely focused on model-free learning methods, often requiring hundreds of years of experience to produce competitive agents. Our approach is based on model-based planning. We tackle the problem of partial observability by first building an (oracle) planner that has access to the full state of the environment and then distilling the knowledge of the oracle to a (follower) agent which is trained to play the imperfect-information game by imitating the oracle's choices. We experimentally show that planning with naive Monte Carlo tree search does not perform very well in large combinatorial action spaces. We, therefore, propose planning with a fixed-depth tree search and decoupled TS for action selection. We show that the planner is able to discover efficient playing strategies in the games of *Clash Royale* and *Pommerman* and the follower policy successfully learns to implement them by training on a few hundred battles.**

***Index Terms*—*Clash Royale*, imperfect-information games, Monte Carlo tree search (MCTS), *Pommerman*.**

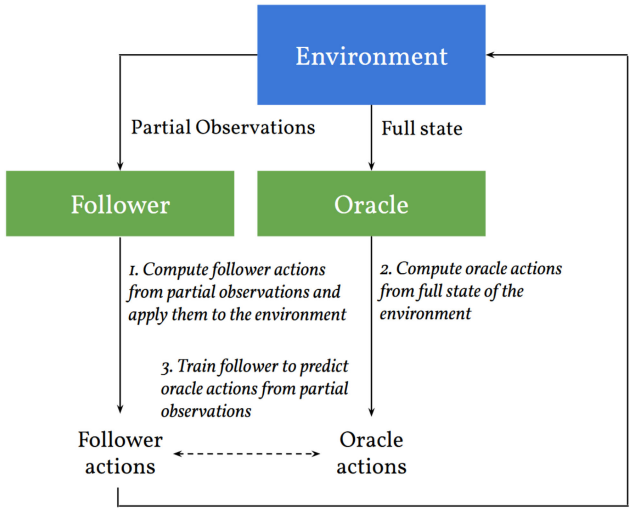

Fig. 1. Proposed approach to solving imperfect-information games.

I. Introduction

THE goal of the field of reinforcement learning (RL) is to develop learning algorithms that can effectively deal with the complexities of the real world. Games are a structured form of interactions between one or more players in an environment, making them ideal for the study of RL. Much of the research in artificial intelligence has focused on games that emulate different challenges of the real world. In *Go* [1], the agent has to discover complex strategies in a large search space. In card games like *Poker* [2]–[4], the agent has to deal with the imperfect-information, such as the unknown cards of the opponent. In *StarCraft II* [5] and *Dota 2* [6], the agent has to compete with other agents who take simultaneous actions from a large action space.

In this work, we consider the problem of learning to play imperfect-information multiplayer games with simultaneous moves and large state-action spaces. We consider two such games as learning environments: *Clash Royale* and *Pommerman* [7]. *Clash Royale* is a popular multiplayer real-time strategy (RTS) game combining elements of different genres such as multiplayer online battle arena, collective-card games, and tower defense games. The complexity in learning to play *Clash Royale* comes from the presence of cyclic strategies, partial observability, and exploration in large dynamic action spaces (more details in Section II-A). *Pommerman* is a popular multiagent RL benchmark which is difficult due to the need for opponent modeling and, therefore, a large branching factor as decisions are made in the combinatorial action space.

In this article, we introduce an algorithm for efficient learning in large imperfect-information games,[1] which does not require modifying the core game implementation. Our approach (illustrated in Fig. 1) consists of two separate components: an oracle planner and a follower agent. The oracle planner has access to the full state of the environment and performs self-play tree search to compute effective (oracle) actions for each player. The oracle planner by itself can be used to implement a cheating AI for game implementations that do not support randomizing hidden information. A follower agent that can play the imperfect-information game is obtained by training a neural network to predict the oracle actions from partial observations using supervised learning.

Manuscript received August 15, 2020; revised December 1, 2020 and February 18, 2021; accepted March 17, 2021. Date of publication March 22, 2021; date of current version June 16, 2022. This work was supported by Supercell Oy, Aalto University, and Academy of Finland (project 314881). *(Corresponding author: Rinu Boney.)*

Rinu Boney, Alexander Ilin, and Juho Kannala are with the Department of Computer Science, Aalto University, 02150 Espoo, Finland (e-mail: rinu.boney@aalto.fi; alexander.ilin@aalto.fi; juho.kannala@aalto.fi).

Jarno Seppänen is with Supercell, 00180 Helsinki, Finland (e-mail: jarno.seppanen@supercell.com).

This article has supplementary material provided by the authors and color versions of one or more figures available at https://doi.org/10.1109/TG.2021.3067723.

Digital Object Identifier 10.1109/TG.2021.3067723

[1] See https://sites.google.com/view/l2p-clash-royale for an explanation video of our method, in the context of *Clash Royale*.

Planning is nontrivial in imperfect-information games [8]. The classical solution is to use Monte Carlo tree search (MCTS) with *determinization* of the hidden information during search to account for the lack of the fully observed state of the environment [9]–[12]. However, this approach cannot be directly used in practice for many games as most existing simulators do not support the possibility of varying the hidden information.

Simultaneous moves with large action spaces makes model-based planning exceptionally challenging. Conventional MCTS can easily get stuck at creating new nodes corresponding to untried actions in a combinatorial action space. In this article, we propose to build an oracle planner based on fixed-depth tree search (FDTS) with the use of decoupled Thompson sampling (TS) for action selection. Our experiments show that FDTS can discover efficient strategies via self-play in the two challenging games that we consider in the article.

Contributions: 1) We introduce an algorithm for efficient planning and learning in large imperfect-information games with implementations that do not support varying of hidden information. 2) We demonstrate that naive MCTS can be problematic in large action spaces and introduce FDTS to improve the quality of planning. 3) We demonstrate the effectiveness of the algorithm in the novel setting of *Clash Royale* and the popular multiagent RL benchmark of *Pommerman*.

II. Imperfect-Information Games

We formalize the imperfect-information games considered in this article as partially observable stochastic games (POSG) [13]. In POSG, a game is played by a set of N players and each game begins in an initial state s_0 sampled from an initial state distribution. In any state s, observation functions $O_i(s)$ yield observations $o_i = O_i(s)$ for each player i. After receiving observation o_i, each player i chooses an action $a_i \in A_i(s)$, where $A_i(s)$ is the set of actions available to player i in state s. Once all players choose actions $a = (a_1, \ldots, a_N)$, the game transitions to a new state s' as defined by a transition function $s' = f(s, a)$. Thus, the joint action space is $A(s) = A_1(s) \times \cdots \times A_N(s)$. The end of a game is defined by a set of terminal states Z. Once the game reaches a terminal state $z \in Z$, all players receive a ternary reward of 1 (win), 0 (draw), or –1 (loss) as defined by a reward function $R_i(z)$. A player does not have access to the true initial state distribution or the transition function but can sample from them by playing games. We now introduce the two games studied in this article.

A. Clash Royale

Clash Royale is a multiplayer RTS game consisting of short battles lasting a few minutes. We focus on the two-player mode of *Clash Royale*. Before a battle, each player picks a *deck* of eight different cards that is not revealed to the opponent. The game has nearly 100 *cards* that represent playable troops, buildings, or spells that will be used in battles. As the game begins, each player is dealt a random subset of four different cards (*hand*) from their deck. Solving the whole game of *Clash Royale* involves solving the meta-game of choosing the right deck. In this article, we focus on a fixed beginner deck consisting of *Knight*, *Giant*, *Archer*, *Arrows*, *Minions*, *Fireball*, *Musketeer*, and *Baby Dragon*.

Fig. 2. Screenshot of *Clash Royale* (left) and *Pommerman* (right).

TABLE I
State Space of a Battle in *Clash Royale*

	Feature	Description
Cards	Decks	Sets of 8 cards chosen by both players for this battle.
	Level	Levels of all cards in the chosen decks.
	Cost	Elixir cost of all cards in the chosen decks.
	Hand	Sets of 4 cards currently available to both players.
	Next card	Next card to be dealt to each player.
Progress	Elixir	Elixir currently available to each player.
	Time	Time elapsed since beginning of battle.
	2x Elixir	Is battle in double elixir mode?
	Overtime	Is battle in sudden death mode?
	Past actions	List of actions by the player in the past 10 steps.
Objects	Type	Types of all objects in the battle arena.
	Position	x, y coordinates of all objects in the arena.
	Level	Levels of all objects in the arena.
	Health	Health of all objects in the arena.
	Color	Object belongs to blue or red player?

Battles in *Clash Royale* are played on a visually immersive 18×32 board initially consisting of a king tower and two princess towers for each player (see Fig. 2). The gameplay primarily consists of players deploying cards from their hand onto the battle arena to destroy the towers of the opponent. Each card has an *Elixir* cost associated with it and a card can only be deployed if the player has enough Elixir. Once a card is deployed in a specific location, it creates a troop or building or spell in the battle arena that follows predefined behaviors, and the player is dealt a new card from the deck. A battle ends instantaneously if a king tower is destroyed. If not, the player with the highest number of towers after 3 min wins. Otherwise, the battle extends for an overtime of 2 min and the first player to destroy an enemy tower wins. Otherwise, the battle results in a draw.

The state s of *Clash Royale* is comprehensively defined in Table I. Each player observes the state of the battle arena, battle progress, the player's own hand, and the next card. Information about the cards of the other player is not visible. At any game state, player i can choose either to deploy a legal card (a card that costs less than or equal to available elixir) or to wait for one time step. In this article, an agent interacts with the *Clash Royale* game engine such that one time step corresponds to 0.5 s. The action a_i of deploying card c by player i can be represented

as a tuple (c, x, y) where c is a card identifier and (x, y) is the deploy position in the discrete 18×32 battle arena. The action of waiting is represented with a special *Wait* card. Additionally, we augment the action space with cards in the hand that are illegal (with not enough Elixir). Choosing an illegal card forces the agent to intentionally wait until that card becomes available, after which it can choose to deploy any legal card or wait further. The action space augmented in this way aids uniform exploration of all cards in the game and we use this in all our experiments.

Although the rules of *Clash Royale* are easy to learn, the game has great depth coming from predicting your opponent's moves, including their predictions of yours, which makes it hard to master. Playing *Clash Royale* effectively requires a well-coordinated combination of attacks and defenses and fast adaptation to the opponents' deck and style of play. Further, because of limited Elixir resources and hidden information, waiting for a good deploy time is an important part of strategy. Below, we describe the various scientific challenges in learning to play *Clash Royale*.

1) Cyclic strategies: The cards in the game are designed such that each card can be countered effectively with another card (that is, *Clash Royale* is a *nontransitive game*). Like the game of rock–paper–scissors, there is no single best deterministic strategy.
2) Partial observability: Cards of the opponent are hidden and are only revealed throughout the opponent's deploys. Players can deceive their opponents by choosing to hide cards (not deploy) until later in the game (akin to bluffing).
3) Exploration: At any time during a battle in *Clash Royale*, only *legal cards* (cards with costs less than the currently available Elixir) can be deployed. Naive exploration methods that choose random actions at each step leads to a greedy strategy of almost always deploying the card with the lowest cost (and thereby depleting Elixir). Good exploration strategies have to intentionally wait for the costlier cards.
4) Dynamic, large, and discrete action space: *Clash Royale* has a large discrete action space with the possibility to deploy any of 100 cards in the 18×32 arena ($\sim$60 000 discrete actions). However, at a particular time in a battle, it is only possible to deploy from the legal cards in the hand.

B. Pommerman

Pommerman is a popular multiagent RL benchmark based on the classic Nintendo game *Bomberman*. Battles in *Pommerman* are played on a 11×11 board initialized randomly with rigid walls and wooden walls (that may contain some power-ups) and four players near each corner (see Fig. 2). The players can move in horizontal or vertical directions (that are not blocked by walls or bombs), collect power-ups, or lay bombs in their current locations. A player dies when they are on a tile affected by a bomb blast and effective gameplay requires strategic laying of bombs to knock down all of the opponents. Hidden information in *Pommerman* consists of power-ups hidden inside wooden walls and the power-ups collected by other players. The *Pommerman* benchmark consists of different scenarios and we consider the free-for-all (FFA) variant in this article. The goal of each agent in the FFA mode is to be the last agent to stay alive within a fixed-length episode of 800 time steps. The challenges in performing tree search on *Pommerman* involves: 1) the large branching factor (up to 1296) caused by four players simultaneously choosing from six actions; 2) the difficulty in credit assignment due to the presence of four players; and 3) the common noisy rewards caused by suicides. To assist learning, we mask out actions that immediately lead players into walls or flames (suicide).

We use a Cython implementation of the *Pommerman* environment based on [14]. For clarity of our experimental setup and ease of reproducibility, we open source the code for our *Pommerman* experiments here: https://github.com/rinuboney/l2p-pommerman.

III. Oracle Planner With Full Observability

In our approach, we first build an oracle planner which has access to the full game state. The goal of planning is to discover the optimal sequence of actions that maximize expected rewards. A dynamic programming approach to the planning problem involves estimating expected rewards for every legal action in each state, after which one can act greedily by choosing the action with the largest expected reward. A policy π_i of player i is a distribution over actions available in state s for player i, that is, $a_i \sim \pi_i(a_i|s)$. Let $\pi(a|s) = \pi_1(a_1|s)\pi_2(a_2|s)$ be the joint policy followed by players $i \in \{1, 2\}$. Let $z \sim p(z|s, \pi)$ be the probability distribution over the set of all terminal states induced by following policy π from state s. The state value function $V_i(s)$ is the mean reward of player i while players follow policy π from state s

$$V_i(s) = \mathbb{E}_{z\sim p(z|s,\pi)}[R_i(z)]. \tag{1}$$

The state-action value function $Q_i(s, a)$ is the mean reward of player i while players first take actions $a = (a_1, a_2)$ and then follow policy π from state s

$$Q_i(s, a) = \mathbb{E}_{z\sim p(z|s,a,\pi)}[R_i(z)]. \tag{2}$$

A possible way to do planning is to estimate $Q_i(s, a)$ for each player and choose the action for each player which maximizes its expected reward. One problem with this approach is that one has to consider all combinations of actions (a_1, a_2), which is prohibitive in games like *Clash Royale* where each player chooses from tens of thousands of actions.

In this article, we take a different approach. We assume that the actions a_1 and a_2 are chosen independently, that is, we estimate $Q_i(s, a_i)$ taking an expectation over the opponent policy

$$Q_i(s, a_i) = \mathbb{E}_{z\sim p(z|s,a_i,\pi)}[R_i(z)]\,. \tag{3}$$

With this approximation, the problem formulation can be seen as a partially observable Markov decision process from the perspective of each player, where the opponent is subsumed into the stochastic environment. At the end of planning, each player independently chooses the action that maximizes the estimated Q values

$$a_i = \underset{a_i\in A_i(s)}{\operatorname{argmax}}\, Q_i(s, a_i)\,.$$

A. Monte Carlo Search

Monte Carlo search (MCS) [15] is a simple search method where $Q_i(s, a_i)$ is estimated for all actions $a_i \in A_i(s)$ by performing several iterations of random *rollouts* from state s. That is, both players estimate $Q_i(s, a_i)$ assuming that policies π_1 and π_2 are uniform distributions over the legal actions in every state. In practice, we perform random rollouts for a fixed number of steps and then use a value function estimate V to evaluate the final state. In each iteration of MCS from state s, both players independently and randomly choose actions $a_i \in A_i(s)$ and continue to do so for a fixed number of steps (planning horizon), to reach state $\tilde{s}$. At the end of an iteration, the estimate of $Q_i(s, a_i)$ is updated based on the value estimate $V(\tilde{s})$.

B. Multi-Armed Bandits

MCS can be improved by exploring more promising actions more often. This can be achieved by viewing action selection as a multiarmed bandit (MAB) problem: In the current state s, player i has to choose an action $a_i \in A_i(s)$ with maximum expected reward. There are $|A_i(s)|$ arms and player i can explore new actions or exploit actions with highest value estimates. When MCS is enhanced by MAB, the MAB selection is done at the current state s and the value estimates $Q_i(s, a_i)$ are obtained as in MCS by performing random rollouts.

In this article, we use a *decoupled* approach to action selection: Each player independently chooses an action $a_i \in A_i(s)$ using its own instance of an MAB; thus, the opponents are subsumed into the stochastic environment. We consider two popular MAB algorithms: the upper confidence bound (UCB) and TS.

1) Upper Confidence Bound: UCB algorithms estimate the UCB that any given action is optimal [1], [16]. While there exist different variations of UCB, we consider the commonly used UCB1 variant introduced in [17]. Each player i independently estimates the upper confidence bound $\mathrm{UCB}_i(s, a_i)$ for each action $a_i \in A_i(s)$ as

$$\mathrm{UCB}_i(s, a_i) = Q_i(s, a_i) + c\sqrt{\frac{\log N}{n_{a_i}}} \tag{4}$$

where the c hyperparameter controls the exploration–exploitation tradeoff, n_{a_i} is the visit counts of action a_i and $N = \sum_{a_i \in A_i(s)} n_{a_i}$.

In each iteration, the action with the highest UCB value is chosen deterministically. At the end of planning, normalized visit counts define a probability distribution over actions. The final action can be chosen stochastically by sampling from this distribution or by deterministically choosing the action with the highest visit count.

2) Thompson Sampling: TS [18] maintains probability distributions of cumulative rewards for each action and chooses actions according to the probability that they are optimal. Since the rewards in *Clash Royale* and *Pommerman* are binary, the probability that taking action a_i will lead to a win can be modeled using the Bernoulli distribution. The mean parameter θ_{a_i} of the Bernoulli distribution can be modeled with a Beta distribution which is the conjugate prior distribution for the Bernoulli likelihood. The parameters of the Beta distribution can be updated by maintaining win and loss counts (S_{a_i} and F_{a_i}, respectively) for each action. Note that this posterior update assumes independent samples from a Bernoulli distribution, even though this is not true in a multiplayer setting. During each iteration of planning, the action is chosen as

$$a_i = \underset{a_i \in A_i(s)}{\operatorname{argmax}}\, \theta_{a_i}$$

$$\theta_{a_i} \sim \mathrm{Beta}(S_{a_i} + \alpha, F_{a_i} + \beta).$$

In all the experiments in the article, we set $\alpha = \beta = 1$ and do not tune these hyperparameters. At the end of planning, the final action can be chosen stochastically in a similar manner or deterministically based on the estimated means of the Beta distributions.

C. Monte Carlo Tree Search

MCS described previously has several limitations: 1) It only plans actions for the current state and hence cannot discover effective action combinations; 2) it discards all information about future states and actions traversed during rollouts and plans from scratch in each step; and 3) the rollout policy is random and hence the estimated Q values are under the assumption that both players will act randomly in the future.

MCTS builds upon MCS by considering action selection in all states encountered during rollouts as a MAB problem. MCTS is a best-first tree search algorithm and begins from a root node corresponding to current state s. We start with the most common variant of MCTS in which each MCTS iteration from current state s consists of the following steps.

1) Selection–expansion: Starting at the root node (which corresponds to the current state of the game), a *tree policy* is used to descend through the tree until a new state s' is reached. In the case of two players acting simultaneously, the tree policy can be implemented by both players independently choosing actions $a_i \in A_i(s)$ using one of the MAB algorithms discussed in Section III-B.
2) Evaluation: The value $V(s')$ of the new state s' is evaluated, which can be done in different ways: 1) by applying a handcrafted or a learned value function to s', 2) by random rollout(s) from state s' until a terminal state z and using $R(z)$ as a Monte Carlo estimate of the value, or 3) by a fixed length rollout and applying a value function to the reached state.
3) Backup: The values $Q_i(s, a_i)$ for all the ancestors of node s' are updated using the estimate $V(s')$ and the visit counts n_{a_i} are incremented by one.

See Fig. 3 for a simplified illustration of one MCTS iteration.

After several planning iterations, both players independently choose their best actions and the search tree built by MCTS is reused for planning in subsequent states by moving the root node to the child node corresponding to the chosen joint action. MCTS allows for discovery of effective sequence of actions, reuse of statistics computed from previous states, and iterative improvement of the rollout policy.

A potential problem with MCTS is that the selection–expansion step may stop very early in the tree. This is likely to happen in the games with a large branching factor of the search tree. It is very probable that the tree policy will encounter a novel game state in one of the upper levels of the tree, after which the state is evaluated. This can limit the effective planning

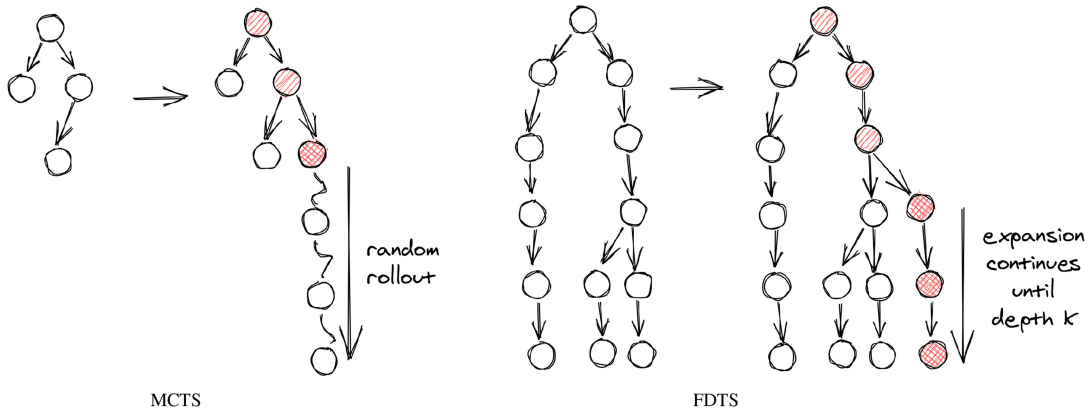

Fig. 3. Illustration of how a search tree is modified in one planning iteration in MCTS (left) and FDTS (right). Nodes visited in the current iteration are shaded. Previously visited nodes are shaded with parallel lines and newly expanded nodes are shaded with a cross-hatch pattern.

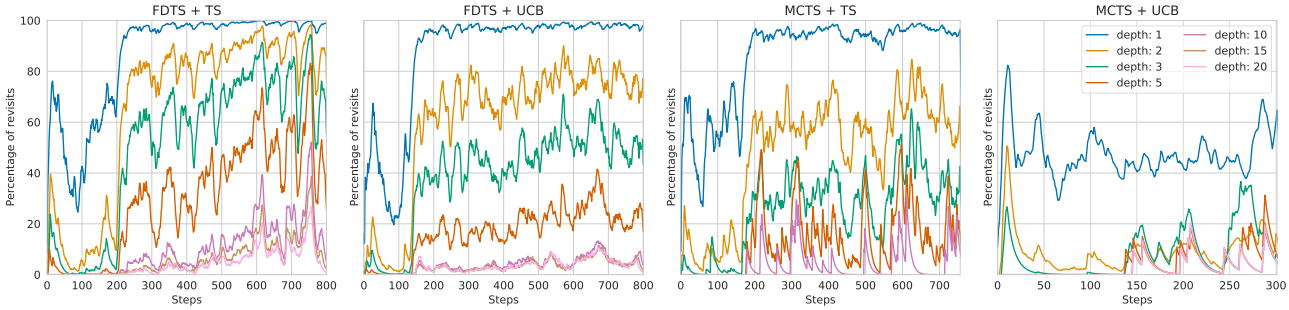

Fig. 4. Comparison of different planners based on the reuse of information stored in the search tree, in a game of *Pommerman*. A *Pommerman* game can last for a maximum of 800 steps and, in each step, we execute the planning procedure for 100 iterations (and a fixed horizon of 20 in the case of FDTS). We plot the (low-pass filtered) ratio of state revisits during planning at each game step (that is, in all of the times, the planner visits a state in depth d of the search tree, the ratio of states that it has previously visited.) We use this to measure the effectiveness of use of information stored in the search tree. The best-performing FDTS+TS planner frequently reuses information, even up to the maximum depth of 20.

horizon of MCTS and makes it problematic to properly evaluate long-term plans.

D. Fixed-Depth Tree Search

We propose to improve MCTS by encouraging planning at least several steps ahead from the current state. The proposed algorithm that we call *FDTS* consists of the following steps.

1) Selection–expansion–rollout: Starting at the root node, an MAB tree policy is applied exactly k times to descend through the tree. If the game reaches a novel state at a particular level, a new node is added to the tree and the tree policy continues action selection from that node until a desired depth level k is reached. These steps result in creating a new branch with a leaf node with state s' at a particular depth level.
2) Evaluation: The value of the node state s' reached at depth k is evaluated. In our experiments, the evaluation step is done by applying a handcrafted value function without performing random rollouts.
3) Backup: The values $Q_i(s, a_i)$ for all the ancestors of node s' are updated using the estimate $V(s')$.

One iteration of FDTS is illustrated in Fig. 3 and the Python pseudocode for FDTS can be found in Listing 1.

The proposed algorithm can be viewed as combining in one step the selection–expansion step and the fixed-length rollout part of the evaluation step of classical MCTS. After a novel state is reached, the MAB algorithm is recursively used to expand that node into a branch that reaches a fixed tree depth k. This is essentially equivalent to a random rollout. The important difference is that we add nodes to the tree for all the states encountered during the random rollout.

Keeping the trajectories encountered during random rollouts may seem wasteful, especially for problems with a large branching factor. However, this turns out to work well in the games considered in this article because the MAB selection process systematically revisits nodes existing in the tree despite the large branching factor. In Fig. 4, we demonstrate that the FDTS equipped with UCB and especially with TS reuses information collected in the previous planning steps. The increased percentage of revisited nodes in FDTS compared to MCTS suggests that storing the rollout trajectories in the search tree is indeed beneficial. The same figure shows that TS tends to revisit existing nodes more often than UCB and this further improves the quality of planning, which is supported by our experimental results.

E. Memory and Computation Requirements

MCS is simple to implement and has minimal memory and computation requirements. MCS only stores statistics of legal actions in the current state. MCTS and FDTS require storing statistics of legal actions in all previously visited states of an

TABLE II
FOUR-PLAYER *POMMERMAN* FFA: PERCENTAGES OF WINS, DRAWS, AND LOSSES AGAINST THREE RULE-BASED OPPONENTS COMPUTED AFTER 400 GAMES

	Rule-based	MCS		MCTS		**FDTS**	
		UCB	TS	UCB	TS	UCB	**TS**
Wins	19.2	35.8	36.3	37.3	40.8	42.3	**51.3**
Draws	23.4	41.0	37.5	32.4	36.1	32.5	**29.0**
Losses	57.4	23.2	26.2	30.3	23.1	25.2	**18.2**

The first column is evaluation of the rule-based agent against three copies of itself.

TABLE III
COMPARISON OF MCTS WITH AND WITHOUT RANDOM ROLLOUTS (UNTIL DEPTH OF 20) IN *POMMERMAN* FFA AGAINST THREE RULE-BASED BASELINE OPPONENTS ON 400 GAMES

	MCTS w/ random rollouts		MCTS w/o random rollouts	
	UCB	TS	UCB	TS
Wins	37.3%	40.8%	21.0%	17.8%
Draws	32.4%	36.1%	17.7%	14.2%
Losses	30.3%	23.1%	61.3%	68.0%

For UCB, we use $c = 2$.

TABLE IV
CLASH ROYALE: COMPARISON OF MCS AGENTS EQUIPPED WITH THREE MAB ALGORITHMS FOR ACTION SELECTION. THE PLANNING HORIZON IS $k = 50$

	Win rate
UCB vs Random	$93.5 \pm 2.4\%$
TS vs Random	$98.5 \pm 1.2\%$
TS vs UCB	$64.0 \pm 4.7\%$

Shown are win rates and 95% confidence intervals.

episode. The main computation in MCS is the stepping forward of the game state using the game engine. MCTS and FDTS further require more computation at every state for action selection using an MAB algorithm.

IV. EXPERIMENTS ON PLANNING WITH THE ORACLE

In this section, we evaluate the proposed planning algorithms on the games of *Pommerman* and *Clash Royale*. Although optimal policies in multiplayer games are stochastic, similar to [19], we observe that deterministic policies perform better in practice. In all the experiments presented in this article, we deterministically choose the action with the highest value at the end of planning.

A. Pommerman

Since *Pommerman* FFA is a four-player game, we compare different planning algorithms by pitting them against three copies of the strong rule-based agent that is provided along with the *Pommerman* environment. It is important to note that the proposed algorithms perform planning in the self-play mode using decoupled action selection for each player, that is, they are not aware of the policy of the rule-based agents. Planning against known agents would be a much easier task.

In *Pommerman*, the number of legal actions for each player can vary from 1 to 6, that is, the branching factor of the search tree can vary from 1 to 1296. In all our experiments, we perform 100 simulations of the planning algorithm at every time step and use a planning depth of $k = 20$ (in MCS and FDTS). In the evaluation step of tree search, we use the reward function of the *Pommerman* (described in Section II) as the value function. This choice was motivated by [14].

In Table II, we report the number of wins, draws, and losses in 400 games for different settings. We consider three planning algorithms, MCS, MCTS, and FDTS, and two alternative ways for actions selection, TS and UCB1 with $c = 2$. For a fair comparison to MCS and FDTS, we use MCTS with random rollouts (at the end of the expansion step in an MCTS iteration, we perform random rollouts till a fixed depth of 20 and use that state for evaluation), which is similar to FDTS except that we do not add the nodes visited during the random rollouts to the search tree. A comparison of MCTS performance with and without this random rollouts is reported in Table III. The best results are obtained with FDTS+TS which attains a win rate of 51.3% with no reward shaping. A similar setup of self-play planning on a Java implementation of the *Pommerman* environment was considered in [20] who reported win rates of 46.5% for MCTS and 33.0% for rolling horizon evolutionary algorithm [21] using shaped rewards.

B. Clash Royale

In *Clash Royale*, the number of discrete actions $A_i(s)$ is very large, but the actions are correlated: deploying a card on nearby positions tend to produce the same outcome. To approximate a good policy, we sample a random set of 64 positions from the space of legal positions for every legal card. A sufficiently large random set would include the optimal deploy positions. With this approximation, in *Clash Royale*, there are two players and the legal actions for each player (with the random sampling of deploy positions) can vary from 1 to 257. That is, the branching factor of the search tree can vary from 1 to 66 049.

In our experiments, we use simple handcrafted value functions for oracle planning: We compute $V(s)$ by doing a rollout from state s assuming that both players do not deploy any more cards. Since the consequences of already deployed cards have predefined behavior, we can reach state s' where the battle arena only contains towers. Then, we evaluate $V(s)$ using the terminal reward function $R(s')$.

We compare UCB1 with $c = 1$, TS and simple random sampling using Monte Carlo search, by pitting one MAB algorithm against another. For example, to compare TS with UCB, Player 1 performs planning using TS for action selection of both players and Player 2 independently performs planning using UCB for action selection of both players. We compute the win rate of an algorithm against another for 400 games in this setting. The results are shown in Table IV. Both TS and UCB clearly outperform random sampling. TS performs the best of all.

While UCB could potentially be fine-tuned to work better with a more comprehensive search over the hyperparameters or using a different UCB variants such as UCB1-Tuned [17], we found TS to robustly work well in most settings. To test the robustness

TABLE V
CLASH ROYALE: WIN RATES OF THOMPSON SAMPLING AGAINST UCB FOR DIFFERENT PLANNING HORIZONS AND UCB EXPLORATION HYPERPARAMETER c

Horizon	c = 0.5	c = 1	c = 2	c = 3	c = 4
10	64.0	69.0	77.6	82.2	82.1
25	81.1	69.1	75.7	83.3	90.7
50	65.6	64.0	75.1	79.8	86.9

Each pair is evaluated on 50 games.

TABLE VI
CLASH ROYALE: COMPARISON OF MCS, MCTS, AND FDTS. THE PLANNING HORIZON IS $k = 50$

	Win rate
MCTS vs MCS	$41.3 \pm 4.5\%$
FDTS vs MCTS	$96.5 \pm 1.8\%$
FDTS vs MCS	$80.3 \pm 3.9\%$

Shown are win rates and 95% confidence intervals.

TABLE VII
CLASH ROYALE: COMPARISON OF MCS WITH MCTS AND FDTS BASED ON THEIR WIN RATES BY EVALUATING EACH PAIR ON 40 GAMES

Horizon	MCTS vs MCS	**FDTS** vs MCS
10	43.7%	**68.1%**
25	19.4%	**70.5%**
50	41.3%	**80.3%**

TABLE VIII
FOUR PLAYER *POMMERMAN* FFA: COMPARISON OF FOLLOWER AGENTS AGAINST THREE RULE-BASED BASELINE OPPONENTS ON 400 GAMES

	Rule-based	**Follower**	
		DAgger	Oracle-behavioral cloning
Wins	19.2%	**23.3%**	17.4%
Draws	23.4%	**22.5%**	19.2%
Losses	57.4%	**54.2%**	63.4%

Evaluation of the rule-based agent against three copies of itself is reported in the first column for reference.

Algorithm 1: Learning to Play Imperfect-Information Games by Imitating an Oracle Planner With DAgger.

```
Initialize follower policy π_f and replay buffer 𝔻.
for each episode do
    Initialize oracle planner π_o.
    for time t until the episode is over do
        Compute follower actions a_f = π_f(o_t) from
        partial observations o_t and apply them to the game.
        Compute oracle actions a_o ~ π_o(s_t) using
        (self-play) tree search, with access to full state s_t.
        Add data (o_t, a_o) to replay buffer 𝔻 and train
        follower policy π_f using 𝔻 to predict the oracle
        actions a_o from partial observations o_t.
    end for
end for
```

of TS, we compare it against UCB with different values of exploration hyperparameter c and planning horizon/depth. The win rates of the comparison in *Clash Royale* is shown in Table V, where TS clearly outperforms UCB. We, therefore, use TS as the MAB algorithm in all our further experiments.

We compare MCS, MCTS, and FDTS in *Clash Royale* by pitting one algorithm against another for 400 games, where each player independently performs planning using the assigned algorithm. The results of our experiments are shown in Table VI. The proposed FDTS planning performs the best.

For further comparison of MCTS and MCS, we pit the two variations of MCTS against MCS for different planning horizons. The win rates on 40 games of *Clash Royale* are shown in Table VII. FDTS outperforms MCS on all planning horizons, with an increased difference for deeper search. These results suggest that FDTS is able to discover better combinations of actions and reuses statistical information (as demonstrated in Fig. 4) to outperform MCS for all planning horizons, with an improved performance as the planning horizon increases.

V. TRAINING FOLLOWER POLICY WITH PARTIAL OBSERVABILITY

Planning enables competitive play with generalization to unseen states. However, the oracle planner has two limitations. 1) It performs many rollouts to make decisions in every state, requiring a game implementation that must run much faster than real time to be able to act in a real-time battle. 2) the oracle planner cheats by having access to the full game state: Private information like the deck and hand of the opponent in *Clash Royale* and hidden power-ups in *Pommerman* becomes visible during future states of planning rollouts. This could be avoided by randomizing hidden information during planning, but the game engines of these games do not support this.

In our approach, we propose to use imitation learning to train a *follower* policy network to perform similarly to the oracle planner but under real-time computation and partial observability. One straightforward way of doing this would be via cloning of the oracle behavior: One can collect trajectories generated by the oracle planner with self-play and use that data to train the follower policy. However, this approach results in a relatively poor performance (see Table VIII).

For better performance, we instead use a version of the DAgger algorithm [22]. In DAgger, the follower policy is trained with the help of the oracle planner in the following way: The trajectories are generated by the follower policy (so that the follower agent visits diverse states), the oracle recommends actions for each state visited by the follower, and the follower policy is updated to predict the recommendations of the expert. In contrast to the original DAgger, we update the policy after every time step instead of after collecting a batch of trajectories. The algorithm for training follower networks is listed in Algorithm 1.

VI. EXPERIMENTS ON TRAINING THE FOLLOWER

We train follower networks to imitate the oracle planner by predicting the oracle action from partial observations o_i. The oracle is chosen to be the best performing FDTS with TS.

TABLE IX
HYPERPARAMETERS OF ORACLE PLANNER AND FOLLOWER NETWORK IN *POMMERMAN*

	Parameter	Values
Oracle	Planning Depth	$[10, 15, \mathbf{20}]$
	Num. Iterations	$[50, \mathbf{100}]$
Follower	Batch Size	$[\mathbf{32}, 64, 128]$
	Learning Rate	$\mathbf{0.001}$

We report all the values considered during random search and the final chosen values are highlighted.

TABLE X
HYPERPARAMETERS OF ORACLE PLANNER AND FOLLOWER NETWORK IN *CLASH ROYALE*

	Parameter	Values
Oracle	Planning Depth	$[10, 25, \mathbf{50}]$
	Num. Iterations	$4\max(\lvert A_1(s)\rvert, \lvert A_2(s)\rvert)$
	Num. Positions	$[32, \mathbf{64}]$
Follower	Batch Size	$[\mathbf{32}, 64, 128]$
	Learning Rate	$[0.001, \mathbf{0.0003}]$
	Embedding Size	$[\mathbf{32}, 64]$
	Hidden Size	$[64, \mathbf{128}, 256]$

We report all the values considered during random search and the final chosen values are highlighted.

A. Pommerman

In *Pommerman*, we train a follower network to imitate the oracle planner on 500 battles. We use the same network architecture as [7]. The observations are represented in a 11×11 spatial representation (corresponding to the 11×11 board in the game) with 14 feature maps. The features represent presence and positions of 10 different objects in the board, bomb blast positions and lifetime and the power-ups collected by the agent. The network architecture consists of four convolutional layers with 32 channels (with ReLU activations) and a final linear layer that predicts the softmax probabilities of the six discrete actions. We used random search to tune the hyperparameters (reported in Table IX) of the oracle planner and the follower policy.

We evaluate the *Pommerman* follower against three rule-based opponents and the results are shown in Table VIII. The follower agent trained with DAgger is able to achieve a win rate 23.3%, outperforming the rule-based agent. Note that previous works have achieved high win rates against the rule-based opponent by directly training against it [14], [23]. Instead, we learn without access to the rule-based agent.

B. Clash Royale

In *Clash Royale*, we train a follower network to imitate the oracle planner on 300 battles. The follower network is a seven-layer convolutional neural network (CNN). We encode the objects in the battle arena using learnable embeddings into 18×32 spatial feature maps as inputs to the CNN. We also encode the battle progress, current cards, and past 10 actions into additional feature maps. The CNN architecture consists of three [3×3 Conv $\rightarrow$ BatchNorm $\rightarrow$ ReLU] blocks and a final 1×1 Conv layer to predict the Q values (means of the Beta distributions) of the spatial deploy positions for all legal cards. We also downsample the features maps by a factor of 2 after the first block and upsample it after the second block so that the CNN outputs 18×32 spatial feature maps. During self-play and evaluation, the follower network deterministically chooses the action with the largest Q value. See https://git.io/Jt6WZ for the PyTorch pseudocode of the follower network and Table X for the corresponding hyperparameters.

We evaluate the follower against three baseline agents: 1) *Random:* a simple uniform random policy; 2) *Q-MC:* a model-free agent trained with Monte Carlo value targets [24]; and 3) *Human-BC:* a strong agent trained to imitate human actions. Deep Q-Network (DQN) [25] was not included in the comparison because it was unstable, most likely due to the large action space and delayed actions.

HUMAN-BC is a very strong baseline: It is a mature agent that has been in production for over a year. That agent was trained using behavioral cloning (supervised learning) to imitate human actions from 76 million frames of human replay data from *Clash Royale*. These replays consisted of games played by humans with a good skill level, all from 4000 trophies and above, and played with a diverse set of decks. The architecture of HUMAN-BC and the training parameters were tuned for metrics like prediction accuracy of deployed cards and their deploy positions. The HUMAN-BC agent consists of two feature extraction networks and an action prediction network. A battle arena feature extraction network embeds the objects (along with their features) in the battle arena in a spatial grid based on their positions and extracts features from the spatial inputs using residual blocks. A battle context feature extraction network extracts battle context features based on cards and battle progress, similar to the follower network architecture, but with a larger network consisting of residual blocks. The battle arena and battle context features are combined using a sum operation and an action prediction network consisting of residual block predicts:

1) when to deploy;
2) card to be deployed;
3) deploy position;
4) value of current state (auxiliary task).

The predicted card is deployed onto the predicted deploy position only if the policy predicts that it should be deployed in the current step.

The win rates of all pairs of agents are presented in Table XI. The Q-MC agent does not perform very well as it is able to beat only the random agent. By analyzing its playing style, one can notice that it tends to learn a particular strategy that is easily predictable by human players. The HUMAN-BC agent is very competitive; the analysis of its gameplays suggests that it is able to use strategies which are common for human players.

The oracle planner beats the other agents almost always, which is natural because it has access to more information. By analyzing its gameplays, we observed that the oracle planner was able to discover effective strategies commonly used by human players.[2] Some of the discovered strategies are as follows.

[2]See https://sites.google.com/view/l2p-clash-royale for our supplementary video including the gameplay videos.

TABLE XI
CLASH ROYALE: COMPARISON OF ALL AGENTS BASED ON WIN RATES (AND 95% CONFIDENCE INTERVALS) OF EACH PAIR EVALUATED ON 100 GAMES

	Random	Q-MC	Human-BC	Follower	Oracle
Random	-	$20.9 \pm 7.9\%$	$1.0 \pm 1.9\%$	$1.4 \pm 2.3\%$	$0.0 \pm 0.0\%$
Q-MC	$79.1 \pm 7.9\%$	-	$14.9 \pm 6.9\%$	$8.6 \pm 4.3\%$	$0.0 \pm 0.0\%$
Human-BC	$99.0 \pm 1.9\%$	$85.1 \pm 6.9\%$	-	$28.6 \pm 8.8\%$	$6.1 \pm 4.7\%$
Follower	$\mathbf{98.6 \pm 2.3\%}$	$\mathbf{91.4 \pm 4.3\%}$	$\mathbf{71.4 \pm 8.8\%}$	-	$\mathbf{6.6 \pm 4.8\%}$
Oracle	$100.0 \pm 0.0\%$	$100.0 \pm 0.0\%$	$93.9 \pm 4.7\%$	$93.4 \pm 4.8\%$	-

Win rates of Q-MC and follower are averaged over win rates of networks trained used 5 different seeds.

1) Groups of troops: The planner is consistently playing high-hitpoint "tank" troops like *Giants*, *Knights*, or *Baby Dragons* in the front and support units like *Musketeers* or *Archers* behind the tank. This is a key strategy for successful attacks which requires coordinating deploys across several time steps.
2) Defense against tanks: When attacked by a single tank unit without support units, the planner deploys high damage per second troops like *Musketeer* or *Minions* to directly and efficiently remove the tank. However, if there are support units behind the tank, then the defending planner typically tries to destroy the support units first to minimize potential tower damage from such more threatening attacks.
3) Hedging: *Clash Royale* games often have pivotal moments where one of the players must decide between two high level strategies: trying to defend against an oncoming attack or hedging bets by skipping defense and launching a similarly powerful attack on the other lane. The planning agent is able to decide to forgo defense and respond with an attack against the other tower.
4) Slowing down attacks: If an attack is approaching but there are no good defense cards in the hand, the planner is able to deflect a threatening attack by deploying a tank like *Giant* to slow down the attack and thus rotating more suitable cards to the hand.
5) Race against time: In the end of the game, when both players are equally close to winning, it is essential to damage the opponent's king tower quicker before the opponent damages yours. In these scenarios, the planner is coordinating all deploys at the king tower, using even weak damage from spells like *Arrows*.

Training the follower with oracle supervision resulted in a follower agent that outperforms the strong HUMAN-BC baseline. Although the follower does not have access to the full game state, it successfully uses the strategies discovered by the oracle, which we observed by analyzing its gameplay.

VII. RELATED WORK

Previous works on RL in games with high-dimensional state-action spaces such as StarCraft II [5], Dota 2 [6], and Honor of Kings [26] have used model-free RL algorithms [27], [28], requiring a large amount of data to learn. We take a model-based planning approach to learn to play imperfect-information games. Previous works have found MCTS to be an effective planning algorithm in various simultaneous-move games with low-dimensional state-action spaces [29]–[31], even though it does not have any theoretical guarantees on achieving optimal play in simultaneous-move or imperfect-information games and can be exploited by regret minimization algorithms [29]. MCTS has been used for planning in imperfect-information games essentially by *determinization* of the hidden information [9]–[11], also known as perfect information Monte Carlo [32]. The determinization technique involves performing several instances of the MCTS procedure with different randomizations of the hidden information and average across the resulting policies. Information set MCTS (IS-MCTS) [12] involves determinization of hidden information in each MCTS iterations to construct a search tree of information sets. MCTS algorithms that use determinization [9]–[12] are not applicable to complex games or real-world problems, where it is not possible to randomize hidden information. In this article, we introduce an algorithm for efficient planning and learning in imperfect-information games by using a function approximator to average across the resulting policies produced by an oracle planner that has access to the hidden information. Even though averaging across different actions computed by the oracle in different states are not optimal, similar to previous works [9]–[11], [33], [34], we found it effective in learning strong policies.

Learning to play card-based RTS games was previously considered in [35] using DQN to learn to select cards and computing the deploy positions in a *post hoc* manner using an attention mechanism, which is suboptimal as the deploy positions are never trained.

Guo *et al.* [36] used imitation learning of an MCTS planner in the simpler single-player setting of Atari games, with full observability and a small number of discrete actions. We show that the naive MCTS used in [36] is problematic in imperfect-information simultaneous-move games with large action spaces and introduce FDTS with TS for better planning.

Combinatorial multiarmed bandit (CMAB) algorithms can be applied in settings where the action space of each player consists of combinations of multiple variables [37]–[39]. For example, in *Clash Royale*, an action consists of a card and the (x and y) deploy position of the card. In this work, we resort to use of MAB algorithms as the combinations of 4 cards and a random sample of 64 deploy positions are limited to only 256 arms. Alternatively, CMAB algorithms can be used for a proper treatment of combinatorial action spaces with very large branching factors [40].

VIII. DISCUSSION

We demonstrate good performance on learning to play in the novel setting of *Clash Royale* and the challenging multiagent RL

benchmark of *Pommerman*. Our approach consists of an oracle planner that has access to the full state of the environment and a follower agent which is trained to play the imperfect-information game by imitating the actions of the oracle from partial observations. We demonstrate that naive MCTS is problematic in high-dimensional action spaces. We show that FDTS and Thompson sampling overcome these problems to discover efficient playing strategies in *Clash Royale* and *Pommerman*. The follower policy learns to implement them from scratch by training on a handful of battles. Our two-step approach can be combined in an iterative fashion by improving the oracle planner using Q estimates from the follower policy.

Online evolutionary planning methods [41] have been shown to fare better than tree-search in some games with large branching factors and investigation of such methods in imperfect-information games is a promising line of future work. Other potential directions of future work include use of regret minimization algorithms [2], [42].

While *Clash Royale* serves as a novel setting of RL research, learned agents also have several use cases in game design. For example, we have the following.

1) Agents can do automated testing of new game content, such as new cards or levels.
2) Agents can be used as practice opponents.
3) New single player games can be designed where humans play against computer agents.
4) Agents can provide assistance to new players during tutorial or unlocking of new cards.

IX. Societal Impacts

The research presented in this article can have an impact on the gaming industry. On the positive side, self-play algorithms can replace handcrafted rules which are widely used for 1) designing bots that play a game in the place of a human and 2) producing game content like *boss levels* (fights against a strong computer-controlled enemy). Designing game-specific rule-based bots is an expensive component of game development, and replacing this component with a general self-play algorithm can have a strong positive impact. Self-play bots can also be easily retrained and used to reduce manual work for game testing, which involves finding bugs and assessing the difficulty levels of a game. On the negative side, in the wrong hands, skillful bots can be used for cheating in the game, which is a major issue in video games, especially in online games [43]–[45]. Bots can be used to cheat by providing an unfair advantage to a player during gameplay. If players cannot know for sure that they are playing against other human opponents on equal grounding, it can erode the trust of the player community toward the game. Similarly to any other RL algorithm, our research results alone are not enough to enable cheating in games in general because the model would have to be first trained against a specific game environment and then integrated into the game software, both of which require low-level access to the game engine. Overall, further research in data-efficient RL will increase the risk of bot misuse in games, but dealing with that is a line of future work.

Appendix
Listing 1
Fixed Depth Tree Search

```
def fdts(root_state, search_tree, n_simulations,
     fixed_depth):
    """
    Perform FDTS from root_state for n_simulations rollouts
      of length fixed_depth.
    search_tree is a dictionary that maps previously
     visited states to independent instances of an MAB
     algorithm for each player.
    """
    if root_state not in search_tree:
        search_tree[root_state] = [
            MAB(root_state.legal_actions(player))
            for player in state.players()
        ]
        # MAB is an implementation of a multi-armed bandit
     algorithm like UCB or Thompson sampling
    for _ in range(n_simulations):
        state = root_state
        search_path = []
        for _ in range(plan_horizon):
            decoupled_mab = search_tree[state]
            # Independently select actions for each player
     using the MAB algorithm
            actions = [
                mab.select() for mab in decoupled_mab
            ]
            state = state.apply(actions)
            search_path.append(decoupled_mab)

            if state not in search_tree:
                search_tree[state] = [
                    MAB(state.legal_actions(player))
                    for player in state.players()
                ]
                # Setting plan_horizon to a large value and
      adding a break statement here would result in the
     standard MCTS algorithm

            if state.is_terminal():
                break

        # Evaluate the final state using a handcrafted
     value function
        values = value_fn(state)

        # Update the statistics of all MAB instances
     visited during this rollout
        for decoupled_mab in search_path:
            for mab, value in zip(decoupled_mab, values):
                mab.update(value)

    # Independently select final actions for each player
    actions = [
        mab.act() for mab in search_tree[root_state]
    ]
    return actions
```

Acknowledgment

The authors would like to thank Steven Spencer, Hotloo Xiranood, Mika Seppä, and everybody else at Supercell for fruitful discussions, comments on the draft of this article, computational infrastructure, manual testing of learned agents, and other support. Rinu Boney has done the work as an Intern at Supercell.

References

[1] D. Silver *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.
[2] M. Moravčík *et al.*, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science*, vol. 356, no. 6337, pp. 508–513, 2017.

[3] N. Brown and T. Sandholm, "Superhuman AI for heads-up no-limit poker: Libratus beats top professionals," *Science*, vol. 359, no. 6374, pp. 418–424, 2018.

[4] N. Brown and T. Sandholm, "Superhuman AI for multiplayer poker," *Science*, vol. 365, no. 6456, pp. 885–890, 2019.

[5] O. Vinyals *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[6] C. Berner *et al.*, "Dota 2 with large scale deep reinforcement learning," 2019, *arXiv:1912.06680*.

[7] C. Resnick *et al.*, "Pommerman: A multi-agent playground," 2018, *arXiv:1809.07124*.

[8] N. Brown and T. Sandholm, "Safe and nested subgame solving for imperfect-information games," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 689–699.

[9] I. Frank and D. Basin, "Search in games with incomplete information: A case study using bridge card play," *Artif. Intell.*, vol. 100, no. 1-2, pp. 87–123, 1998.

[10] M. L. Ginsberg, "Gib: Imperfect information in a computationally challenging game," *J. Artif. Intell. Res.*, vol. 14, pp. 303–358, 2001.

[11] R. Bjarnason, A. Fern, and P. Tadepalli, "Lower bounding Klondike solitaire with Monte-Carlo planning," in *Proc. 19th Int. Conf. Automated Plan. Scheduling*, 2009, pp. 26–33.

[12] P. I. Cowling, E. J. Powley, and D. Whitehouse, "Information set monte carlo tree search," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 2, pp. 120–143, Jun. 2012.

[13] E. A. Hansen, D. S. Bernstein, and S. Zilberstein, "Dynamic programming for partially observable stochastic games," in *Proc. AAAI*, vol. 4, 2004, pp. 709–715.

[14] T. Matiisen, "Pommerman baselines," 2018. [Online]. Available: https://github.com/tambetm/pommerman-baselines

[15] T. Anthony, R. Nishihara, P. Moritz, T. Salimans, and J. Schulman, "Policy gradient search: Online planning and expert iteration without search trees," in *Proc. NeurIPS Deep Reinforcement Learn. Workshop*, 2019.

[16] C. B. Browne *et al.*, "A survey of monte-carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[17] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Mach. Learn.*, vol. 47, no. 2-3, pp. 235–256, 2002.

[18] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, no. 3/4, pp. 285–294, 1933.

[19] P. Perick, D. L. St-Pierre, F. Maes, and D. Ernst, "Comparison of different selection strategies in monte-carlo tree search for the game of tron," in *Proc. IEEE Conf. Comput. Intell. Games*, 2012, pp. 242–249.

[20] D. Perez-Liebana, R. D. Gaina, O. Drageset, E. Ilhan, M. Balla, and S. M. Lucas, "Analysis of statistical forward planning methods in Pommerman," in *Proc. AAAI Conf. Artif. Intell. Interactive Digit. Entertainment*, vol. 15, no. 1, 2019, pp. 66–72.

[21] D. Perez, S. Samothrakis, S. Lucas, and P. Rohlfshagen, "Rolling horizon evolution versus tree search for navigation in single-player real-time games," in *Proc. 15th Annu. Conf. Genet. Evol. Computation*, 2013, pp. 351–358.

[22] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *Proc. 14th Int. Conf. Artif. Intell. Statist.*, 2011, pp. 627–635.

[23] C. Gao, P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "Skynet: A top deep RL agent in the inaugural Pommerman team competition," 2019, *arXiv:1905.01360*.

[24] A. Amiranashvili, A. Dosovitskiy, V. Koltun, and T. Brox, "TD or not TD: Analyzing the role of temporal differencing in deep reinforcement learning," in *Proc. Int. Conf. Learn. Representations*, 2018.

[25] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, 2015, Art. no. 529.

[26] D. Ye *et al.*, "Mastering complex control in MOBA games with deep reinforcement learning," 2019, *arXiv:1912.09729*.

[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*.

[28] L. Espeholt *et al.*, "IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures," in *Proc. Int. Conf. Mach. Learn.*, 2018, pp. 1406–1415.

[29] M. Shafiei, N. Sturtevant, and J. Schaeffer, "Comparing UCT versus CFR in simultaneous games," *IJCAI Workshop Gen. Game Playing*, 2009.

[30] F. Teytaud and O. Teytaud, "Lemmas on partial observation, with application to phantom games," in *Proc. IEEE Conf. Comput. Intell. Games*, 2011, pp. 243–249.

[31] B. Bošanskỳ, V. Lisỳ, M. Lanctot, J. Čermák, and M. H. Winands, "Algorithms for computing strategies in two-player simultaneous move games," *Artif. Intell.*, vol. 237, pp. 1–40, 2016.

[32] J. R. Long, N. R. Sturtevant, M. Buro, and T. Furtak, "Understanding the success of perfect information Monte Carlo sampling in game tree search," in *Proc. 24th AAAI Conf. Artif. Intell.*, 2010, pp. 134–140.

[33] J. Schöfer, M. Buro, and K. Hartmann, "The UCT algorithm applied to games with imperfect information," Diploma thesis, Otto-Von-Guericke Univ. Magdeburg, Magdeburg, Germany, Jul. 11, 2008.

[34] M. Buro, J. R. Long, T. Furtak, and N. Sturtevant, "Improving state evaluation, inference, and search in trick-based card games," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, 2009, pp. 1407–1413.

[35] T. Liu, Z. Zheng, H. Li, K. Bian, and L. Song, "Playing card-based RTS games with deep reinforcement learning," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, 2019, pp. 4540–4546.

[36] 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 *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 3338–3346.

[37] Y. Gai, B. Krishnamachari, and R. Jain, "Learning multiuser channel allocations in cognitive radio networks: A combinatorial multi-armed bandit formulation," in *Proc. IEEE Symp. New Front. Dyn. Spectr.*, 2010, pp. 1–9.

[38] W. Chen, Y. Wang, and Y. Yuan, "Combinatorial multi-armed bandit: General framework and applications," in *Proc. Int. Conf. Mach. Learn.*, 2013, pp. 151–159.

[39] S. Ontanón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *Proc. 9th AAAI Conf. Artif. Intell. Interactive Digit. Entertainment*, 2013, pp. 58–64.

[40] S. Ontanón, "Combinatorial multi-armed bandits for real-time strategy games," *J. Artif. Intell. Res.*, vol. 58, pp. 665–702, 2017.

[41] N. Justesen, T. Mahlmann, S. Risi, and J. Togelius, "Playing multiaction adversarial games: Online evolutionary planning versus tree search," *IEEE Trans. Games*, vol. 10, no. 3, pp. 281–291, Sep. 2018.

[42] M. Zinkevich, M. Johanson, M. Bowling, and C. Piccione, "Regret minimization in games with incomplete information," in *Proc. Adv. Neural Inf. Process. Syst.*, 2008, pp. 1729–1736.

[43] J. Blackburn, N. Kourtellis, J. Skvoretz, M. Ripeanu, and A. Iamnitchi, "Cheating in online games: A social network perspective," *ACM Trans. Internet Technol.*, vol. 13, no. 3, pp. 1–25, 2014.

[44] X. Zuo, C. Gandy, J. Skvoretz, and A. Iamnitchi, "Bad apples spoil the fun: Quantifying cheating in online gaming," in *Proc. 10th Int. AAAI Conf. Web Social Media*, 2016, pp. 496–505.

[45] J. Paay, J. Kjeldskov, D. Internicola, and M. Thomasen, "Motivations and practices for cheating in Pokémon Go," in *Proc. 20th Int. Conf. Hum.-Comput. Interaction Mobile Devices Serv.*, 2018, pp. 1–13.