Past, Present, and Future: An Optimal Online Algorithm for Single-Player GDL-II Games

Florian Geißer and **Thomas Keller** and **Robert Mattmüller**[1]

Abstract. In General Game Playing, a player receives the rules of an unknown game and attempts to maximize his expected reward. Since 2011, the GDL-II rule language extension allows the formulation of nondeterministic and partially observable games. In this paper, we present an algorithm for such games, with a focus on the single-player case. Conceptually, at each stage, the proposed NORNS algorithm distinguishes between the past, present and future steps of the game. More specifically, a belief state tree is used to simulate a potential past that leads to a present that is consistent with received observations. Unlike other related methods, our method is asymptotically optimal. Moreover, augmenting the belief state tree with iteratively improved probabilities speeds up the process over time significantly.

As this allows a true picture of the present, we additionally present an optimal version of the well-known UCT algorithm for partially observable single-player games. Instead of performing hindsight optimization on a simplified, fully observable tree, the true future is simulated on an action-observation tree that takes partial observability into account. The expected reward estimates of applicable actions converge towards the true expected rewards even for moves that are only used to gather information. We prove that our algorithm is asymptotically optimal for single-player games and POMDPs and support our claim with an empirical evaluation.

1 INTRODUCTION

Games have played an important role in AI research since the early days of the field. Nowadays, state-of-the-art players for games like chess and checkers are able to defeat humans on grandmaster level [2, 11]. Even for more complex games like Go [5] and for partially observable nondeterministic games like Poker [10], computer players capable of playing on a professional level have been developed. One point of criticism is that all these players strongly depend on game specific knowledge and game specific expertise of their developers. *General game playing* (GGP) is the field concerned with designing players that are able to play games previously unknown to them given only a formal description of the rules of the game. The field has gained more and more attention in the past years, thanks to the annual international GGP competition [6]. In classic GGP, a player receives the rules of a finite, discrete, deterministic, fully observable game encoded in the Game Description Language (GDL) [8] and has a given amount of time per round in order to decide which action to play. Nowadays, most state-of-the-art GGP players use the UCT algorithm [7] to tackle this problem [4, 9]. In 2011, Thielscher [15] introduced the extension GDL-II, GDL with imperfect information, to describe partially observable nondeterministic games. The first approach to the problem of efficiently playing such games was based on treating the game as one with perfect information and applying classic UCT search [12]. Whereas in that work complete belief states were computed, Edelkamp et al. [3] proposed an algorithm that only computes partial belief states. Schofield et al. [13] pointed out the problem of treating the game as a classic GGP game. When using hindsight optimization, information gathering moves are considered worthless because the game is treated as fully observable. They extend their original hyperplay approach by using nested players to perform imperfect information simulations.

In the related field of research in Partially Observable Markov Decision Processes (POMDPs), Silver and Veness [14] present an online planning algorithm that uses a partial-observation variant of the UCT algorithm. Their approach to simulating the past yields an approximation of the current belief state with a particle filter. Even though their method is asymptotically optimal as well, it uses domain specific knowledge and thus cannot be used in the scenario considered here. Our approach has nevertheless been inspired by Silver and Veness. We also simulate future plays with UCT on action-observation trees in order to tackle the problem of hindsight information. We extend their work by a domain independent method that simulates the past to compute the full belief state iteratively, which is based on Monte Carlo belief state tree simulation [3]. We augment the trees with probabilities to generate reasonable presents at any time and to update beliefs efficiently using Bayes' rule.

This paper is structured as follows: We start by giving formal definitions of games, game trees and belief states. Subsequently, we present the procedures that constitute the NORNS algorithm, i.e., the part that simulates what happened in the past and the part that reasons over what will happen in the future. We start by introducing the UCT algorithm on action-observation trees, followed by an introduction of belief state trees, Monte Carlo belief state tree search and Bayesian weight updates. We conclude that section with an overview of the interaction between the different mechanisms. The sections that follow consist of a proof that the NORNS algorithm is asymptotically optimal and an empirical evaluation of the NORNS algorithm on several single-player games. We discuss what is necessary to apply the NORNS algorithm to multi-player games, as well as several challenges which we have yet to overcome.

2 BACKGROUND

A *partially observable nondeterministic game*, or simply a *game*, is a tuple $G = \langle P, S, A, L, \alpha, s_0, S_\star, R, \omega \rangle$ where $P = \{1, \ldots, n\}$ is the set of *players*, S is a finite set of *states*, and A is a finite set of *actions*. For an *action profile* $a = (a_1, \ldots, a_n) \in A^n$, by

[1] University of Freiburg, Germany, email: {geisserf, tkeller, mattmuel}@informatik.uni-freiburg.de

a_p we denote the action of player p, and by a_{-p} the action profile $(a_1, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n)$ of the remaining players. Moreover, $L : P \times S \to 2^A$ defines the set of *legal actions* $L(p, s) \subseteq A$ of player $p \in P$ in state $s \in S$, and $\alpha : S \times A^n \hookrightarrow S$ is a partial function from states and action profiles to successor states, the *successor function*, such that $\alpha(s, a)$ is defined iff $a_p \in L(p, s)$ for all $p \in P$. The state $s_0 \in S$ is the *initial state* of G, and $S_\star \subseteq S$ is the set of *terminal states*. We assume that $L(p, s_\star) = \emptyset$ for all $s_\star \in S_\star$. The *reward function* $R : P \times S_\star \to [0, 1]$ associates a reward $R(p, s)$ with every player $p \in P$ and every terminal state $s \in S_\star$. Since we examine games with partial observability, we call sets of states a player considers possible *belief states* $B \subseteq S$. We denote the set of all belief states with $\mathcal{B} = 2^S$. The initial belief state is $B_0 = \{s_0\}$, and later belief states are the result of applying the *observation function* $\omega : P \times S \times A^n \hookrightarrow \mathcal{B}$, where $\omega(p, s, a)$ is defined iff $\alpha(s, a)$ is defined. The set of legal actions for player p in belief state B is $L(p, B) = \bigcap_{s \in B} L(p, s)$.

Every game $G = \langle P, S, A, L, \alpha, s_0, S_\star, R, \omega \rangle$ induces a game tree of histories starting in the initial state and ending in a terminal state. Formally, the game tree induced by G is the tuple $\mathcal{G}(G) = \langle P, H, \beta, H_\star, R \rangle$, where the set of *players* P is the same as in G, and the set of *histories* H and *states associated with histories* $\beta(h)$ are defined as follows using mutual recursion: the *empty history* $\langle\rangle$ is in H, and the state associated with the empty history is the initial state, i.e., $\beta(\langle\rangle) = s_0$. For each history $h \in H$, also $h' = \langle h, a \rangle \in H$, if $a \in A^n$ and $s' = \alpha(\beta(h), a)$ is defined. Then, $\beta(h') = s'$. Nothing else is a history. The set of *terminal histories* $H_\star \subseteq H$ is the set of histories $h_\star$ with $\beta(h_\star) \in S_\star$. Notice that we do not consider infinite histories here, since we are only interested in finite-horizon games. Finally, the *reward function* $R : P \times H_\star \to [0, 1]$ assigns to a terminal history the reward associated with the corresponding terminal state, i.e., by abuse of notation, $R(p, h_\star) = R(p, \beta(h_\star))$. For sequences $h = \langle x_1, \ldots, x_k \rangle$ and $j \leq k$, we write $h_{\leq j}$ for $\langle x_1, \ldots, x_j \rangle$. In the following, we assume that there exists a horizon $k \in \mathbb{N}$ such that all histories in H have length at most k. Thus every play will lead to a terminal state after at most k steps.

3 The NORNS algorithm

We present the NORNS algorithm for the special case of single-player games, however, we will still have to take a second player, the *random player*, into account. This player is used to model nondeterminism and chooses uniformly between his legal actions, e.g., distributions of cards or results of coin flips. Given an observation, our goal is to choose the best possible action available.

In any given step, the NORNS algorithm distinguishes between three different stages of a game: the *past*, the *present* and the *future*.[2] The past consists of previously performed actions and observations resulting from them, as well as the (unknown) actions of the random player. The algorithm uses belief state tree search [3] to sample a possible present state according to the current belief. Starting from this state, a possible future play is simulated, by performing UCT search on a so-called action-observation tree.

3.1 Action-Observation Trees and UCT Search

Previous approaches to general game playing with imperfect information rely on hindsight optimization for their move selection [12, 3], which leads to the aforementioned problems. Instead of using a vanilla state-action tree representation, we introduce action-observation trees, which allow us to couple actions to belief states.

The *action-observation tree* (AOT) induced by game G for player $p \in P$ is the tuple $AO(G, p) = \langle \mathcal{H}, \gamma \rangle$ where $\mathcal{H}$ is the set of *action-observation histories* of the form $\langle a^1, obs^1, \ldots, a^i \rangle$ or $\langle a^1, obs^1, \ldots, a^i, obs^i \rangle$, i.e., an action-observation history is an alternating sequence of actions and observations. We call the set of action-observation histories $\mathcal{A} = \{h \in \mathcal{H} \mid |h| \text{ odd}\}$ ending in an action the set of *action nodes*, and those ending in an observation, $\mathcal{O} = \{h \in \mathcal{H} \mid |h| \text{ even}\}$, the set of *observation nodes* in the tree. We can formally define $\mathcal{H}$ and *belief states associated with action-observation histories* $\gamma(h)$ by mutual recursion: the empty sequence $\langle\rangle$ is in $\mathcal{H}$ and the belief state associated with the empty sequence is the initial belief state $\gamma(\langle\rangle) = B_0$. For each observation node h, also $h' = \langle h, a_p \rangle \in \mathcal{H}$, if $a_p \in L(p, \gamma(h))$. Then $\gamma(h') = \{\alpha(s, (a_p, a_{-p})) \mid s \in \gamma(h) \text{ and } a_{-p} \in L_{-p}\}$, where $L_{-p} = \prod_{p' \in P \setminus \{p\}} L(p', \gamma(h))$. Intuitively, for each legal action profile a_{-p} of the players other than p and for each state s considered possible in h, $\gamma(\langle h, a_p \rangle)$ contains the successor state obtained by applying action profile (a_p, a_{-p}) in s. In the subsequent observation node layer, the action-observation tree will branch over possible observations, i.e., for each observation node h and following action node $h' = \langle h, a_p \rangle$, also $h'' = \langle h', obs \rangle \in \mathcal{H}$, for all $obs = \omega(p, s, (a_p, a_{-p}))$ for any $s \in \gamma(h)$ and $a_{-p} \in L_{-p}$ as above. Then, $\gamma(h'') = \gamma(h') \cap obs$. Intuitively, this means that for each possible observation obs (for each state and each opponent action profile), there is a new observation node where observation obs is incorporated into the current belief. Nothing else is in $\mathcal{H}$.

To determine the action with the highest expected reward, given an observation, the algorithm uses UCT search on action-observation trees, which simulates future plays, based on the present state of the game. A visit counter $V(h)$ on the nodes of the UCT tree is introduced, as well as a value counter $Q(A)$ on the action nodes which stores the average reward of an action node A.

Initially, the UCT tree contains only the empty history $\langle\rangle$. It then gradually builds up an unbalanced tree which asymptotically approaches the action-observation tree. The algorithm is divided into different stages. Starting from some initial observation obs and some state $s \in obs$ (which is determined by the belief state tree search that is presented in the next subsection), it selects an action a_p that maximizes the UCB1 formula [1] and selects a random action a_{-p} for the random player to compute $obs' = \omega(p, s, (a_p, a_{-p}))$ and $s' = \alpha(s, (a_p, a_{-p}))$. It sets $s := s'$ and $obs := obs'$ and continues this process until $s' \in S_\star$. If there is no node in the tree representing (a_p, obs'), the tree is expanded by adding corresponding action and observation nodes and a simulation with random legal moves for every player is performed until a terminal state $s_\star$ with reward $R(p, s_\star)$ is reached. For every node h visited during the selection, $V(h)$ is incremented, and for every action node A, $Q(A)$ is updated by extending the current estimate with the latest result. Given enough rollouts, the action node values asymptotically approach the expected reward one gets when performing the corresponding action after receiving the observation of the corresponding observation node. In the following subsection, we introduce the algorithm that computes a possible state of the present game, based on past actions and observations.

3.2 Belief State Trees

Belief state trees (BSTs) were introduced by Edelkamp et al. [3] and are a compact representation of the states a player regards possible

[2] In Norse mythology the Norns are female beings who rule the destiny of gods and men, and the three most important of them are associated with the past, the present and the future.

given his previously performed actions and received observations. In order to give a formal definition of such a tree, we need to distinguish between *global histories* and *local histories*. For every global history $h = \langle a^1, \ldots, a^k \rangle \in H$, the *local history of player p* is the sequence of action-observation pairs $h^p = \langle (m_1, obs_1), \ldots, (m_k, obs_k) \rangle$, such that $m_i = a^i_p$ and $obs_i = \omega(p, \beta(h_{\leq i-1}), a^i), i = 1, \ldots, k$. Basically, the local history contains all a player knows about the global history, i.e., his own performed actions and obtained observations. Given such a local history, we can now define a belief state tree as a horizon-limited game tree, together with a marking function that marks possible states of the game, based on the local history of player p. More formally, given a local history $h' = \langle (m_1, obs_1), \ldots, (m_k, obs_k) \rangle$, a belief state tree is a tuple $BST_{h'} = \langle \mathcal{G}_k(G), M \rangle$, where $\mathcal{G}_k(G)$ is the game tree $\mathcal{G}(G)$ with all histories cut off after k steps and $M : H \to \{0, 1\}$ is defined as $M(\langle\rangle) = 1$ and for all $h \neq \langle\rangle$, $M(h) = 1$ iff $h'_{\leq |h^p|} = h^p$ and additionally, for all $a \in A^n$, $\langle h, a \rangle \notin \mathcal{G}_k(G)$ or there exists an $a \in A^n$ with $M(\langle h, a \rangle) = 1$. We call a history marked with 1 a *possible history*. Intuitively, a history is possible iff its corresponding local history for player p is consistent with h' and if all of its successors lie in the future (and therefore not in $\mathcal{G}_k(G)$) or if at least one of its successors is possible. Finally, we call a state s *possible*, if there exists a history h with $M(h) = 1$ and $\beta(h) = s$. If h^τ is the *true global history*, i.e., consists of the true actions of all players, then we also get a unique *true local history* $h^{\tau p}$ and we can then define the unique belief state tree based on this history. The states we consider possible in the current game play are then exactly the possible states on the k-th layer of $BST_{h^{\tau p}}$. If p is clear from the context, we omit p and just write h^τ instead of $h^{\tau p}$.

3.2.1 Belief State Trees Augmented with Probabilities

In reality there often exist actions that are clearly less valuable for a player and therefore states that are possible, but not as likely as some other states. To deal with this matter, given a belief state tree BST, for every history $h \in H$, the algorithm keeps track of a probability distribution P_h over the children of h, where initially $P_h(\langle h, a \rangle) = \prod_{p \in P} P^p_h(a_p)$, with $P^p_h(a_p)$ being the probability that player p plays action a_p in history h. In our setting, $p \in \{us, rnd\}$. Notice that since a belief state tree is used to simulate the past, the algorithm knows exactly which action $\hat{a}_{us}$ it played. Therefore, $P^{us}_h(a_{us}) = 1$ if $a_{us} = \hat{a}_{us}$, and $P^{us}_h(a_{us}) = 0$ otherwise. For the random player uniform action probabilities are assumed.

3.2.2 Monte-Carlo Belief State Tree Search

Monte-Carlo Belief State Tree Search [3] computes a possible state for the current step of the game by computing the marking of BST_{h^τ} on the fly. The basic algorithm, given in Algorithm 1, begins its search at the root of the belief state tree, chooses one random child c and compares the local history for c with the true local history, i.e., tests if the actions for player p in c are consistent with the actions that p really performed and if the observations that p could see are consistent with his real observations. If there is an inconsistency, c is marked as impossible and impossibility markings are propagated upwards in the tree, marking predecessors as impossible if all of their children are marked as impossible. In this case, the belief state tree search for a representative state of the current belief state is restarted. Otherwise, if c is consistent, the search continues with one of its children until the $|h^\tau|$-th (the last) layer of BST_{h^τ} is reached. The state represented by the reached history is then returned.

Algorithm 1 Belief State Tree Search

```
1:  function BSTSEARCH(h^τ):
2:      h ⟵ ⟨⟩
3:      while h has children do
4:          h ⟵ CHOOSECHILD(h)
5:          if h^p is consistent with h^τ, i.e., h^τ_{≤|h^p|} = h^p then
6:              M(h) ⟵ 1
7:          else
8:              M(h) ⟵ 0
9:              while h_{≤|h|−1} has no possible child do
10:                 h ⟵ h_{≤|h|−1}
11:                 M(h) ⟵ 0
12:             BSTSEARCH(h^τ)
13:     return β(h)
```

Notice that for each h^p, consistency with $h^\tau_{\leq |h^p|}$ has to be tested only once. We also do not have to compute the whole, potentially large, belief state. With one call of Algorithm 1, exactly one state from the belief state is computed, which fits well into the UCT search for future plays. Furthermore, we can easily include our probabilistic augmentation. Instead of marking a node as impossible, the algorithm removes it and updates the probabilities of the tree according to Bayes' rule.

3.2.3 Bayes Update

For ease of notation we write $e = h_{\leq |h|-1}$ for the parent of history h if h is clear from the context. Let $c \in H$ be the node we want to remove after detecting it to be inconsistent with the true history. Let $Pred(c) = \{c_{\leq 0}, \ldots, c_{\leq |c|-1}\}$ be the predecessors of c. Let P be the global probability distribution over the leaf nodes of the BST induced by the local probability distributions $P_e(h)$. For a history h, we write h for the event of reaching a leaf node below h. The probability of that event is the sum of the probabilities of all leaf nodes that are reachable from h, i.e. $P(\mathsf{h}) = \sum_{\langle h, \ldots, a_k \rangle \in H} P(\langle h, \ldots, a_k \rangle)$, with k being the last layer of the tree. Since we represent P compactly using local probability distributions P_e over immediate child nodes at all interior nodes e, we can also write $P_e(h) = P(\mathsf{h})/P(\mathsf{e})$. Then, for each history h, $P(\mathsf{h})$ is the product of the local probabilities on the path from the empty history to h, i.e., $P(\mathsf{h}) = \prod_{i=0}^{|h|-1} P_{h_{\leq i}}(h_{\leq i+1})$. Let us assume node c was removed from the tree and we want to update the weight of node h. That means we have to compute $P'_e(h) = P(\mathsf{h}|\neg\mathsf{c})/P(\mathsf{e}|\neg\mathsf{c})$. In words, we want to know the probability of reaching a leaf node below h, given that we are in e and do not pass through c on the way to a leaf node. This can be derived as follows:

$$P'_e(h) = \frac{P(\mathsf{h}|\neg\mathsf{c})}{P(\mathsf{e}|\neg\mathsf{c})} = \frac{P(\neg\mathsf{c}|\mathsf{h}) \cdot P(\mathsf{h}) \cdot P(\neg\mathsf{c})}{P(\neg\mathsf{c}|\mathsf{e}) \cdot P(\mathsf{e}) \cdot P(\neg\mathsf{c})}$$
$$= \frac{P(\neg\mathsf{c}|\mathsf{h})}{P(\neg\mathsf{c}|\mathsf{e})} \cdot P_e(h) = \frac{1 - P(\mathsf{c}|\mathsf{h})}{1 - P(\mathsf{c}|\mathsf{e})} \cdot P_e(h)$$

using Bayes' rule, the definition of $P_e(h)$, and the converse probability, respectively. If $e \notin Pred(c)$, then $P(\mathsf{c}|\mathsf{e}) = 0$. Otherwise, $P(\mathsf{c}|\mathsf{e}) = \prod_{i=|e|}^{|c|-1} P_{c_{\leq i}}(c_{\leq i+1})$, i.e., the product of the local edge weights along the path from e to c. Similarly, if $h \notin Pred(c)$, then $P(\mathsf{c}|\mathsf{h}) = 0$, and otherwise, $P(\mathsf{c}|\mathsf{h}) = \prod_{i=|h|}^{|c|-1} P_{c_{\leq i}}(c_{\leq i+1})$. Clearly, we only have to update the predecessors of c and their children (excluding c), whereas for all other edges, the old edge weight $P_e(h)$ is preserved, i.e., $P'_e(h) = P_e(h)$.

Another interesting feature is that the factors in the product in the numerator in $P'_e(h)$ are a subset of the factors in the product in the denominator. This means the algorithm only has to compute the product of the probabilities from the root to c once and can use them afterwards for every node weight update.

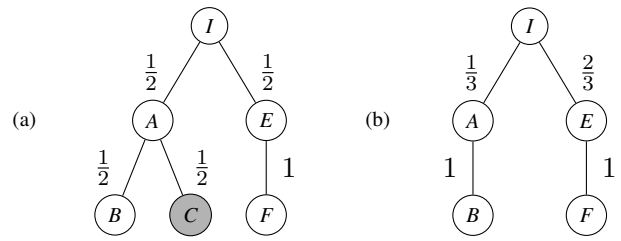

Figure 1: Belief state tree with probabilities; action profiles omitted. (a) Before removal of node C. (b) After removal of node C.

Example 1. Consider the tree in Figure 1(a). We want to remove node C, therefore we need to update the weights of nodes A, B and E. The result is shown in Figure 1(b). The updated local weights are calculated as follows:

$$P'_A(B) = \frac{1-0}{1-1/2} \cdot 1/2 = 1$$

$$P'_I(A) = \frac{1-1/2}{1-(1/2 \cdot 1/2)} \cdot 1/2 = \frac{1/2}{3/4} \cdot 1/2 = \frac{1}{3}$$

$$P'_I(E) = \frac{1-0}{1-(1/2 \cdot 1/2)} \cdot 1/2 = \frac{1}{3/4} \cdot 1/2 = \frac{2}{3} \qquad \square$$

Now that we have defined the algorithms for the different stages of the game, we can put them all together. Let us assume a game is already in progress, i.e., the random player has chosen an action, unknown to the player, in each step of the game and he received corresponding observations. Consider for example Figure 2. The left-hand side shows the BST search, which simulates the past up to the current step, such that it is consistent with own actions and observations. The last performed move was action x with the observation that the random player played action b, resulting in state s. If required the algorithm would have pruned the BST tree and updated its probabilities, which are omitted from the figure.

Starting from s, the UCT algorithm simulates a future play (right-hand side of Figure 2). The observation node o_b is chosen, which corresponds to the observation that the random player played b, and the UCT search is started beginning from o_b, with state s. Own actions are selected according to the UCT selection, actions for the random player are selected uniformly. The tree is traversed until a leaf node is reached. The leaf node is expanded, the game is simulated to the end and the resulting values get propagated back. This algorithm scheme is run until the player has to select the next move he wants to play. Then the move whose corresponding action node has the highest Q value is chosen. Notice that we can prune the UCT tree by removing nodes whose actions lie in the past, i.e., we only need to keep the subtree beginning from o_b.

4 THEORETICAL EVALUATION

In the following, we give a proof sketch that our algorithm asymptotically converges to the optimal action to play in each step of the game. The proof consists of two parts. First, we show convergence of the belief state tree search algorithm to correctly reflecting state probabilities, given enough iterations. Second, we show the correctness of our UCT algorithm.

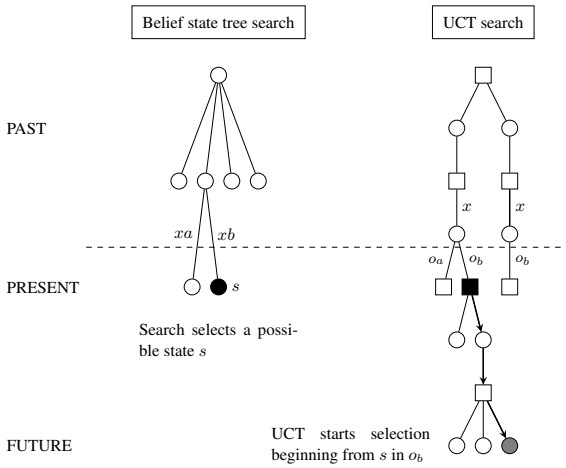

Figure 2: The two steps of the NORNS algorithm

Belief State Tree Search. To prove that our belief update computes the correct belief state including correct probabilities, we show two things. First, we claim that it does not matter if we update the whole tree at once, i.e. we delete all impossible nodes at the same time and update the probabilities afterwards, or we update the tree step by step, i.e. we remove one node from the tree and update the probabilities afterwards. This is simple arithmetic (substitute formulae of later probability updates with formulae of the earlier probability updates, equate with formula which is used if all nodes are updated at once). Second, with enough time, the whole tree will be updated and we show that the NORNS belief update computes the same belief state as the belief update on the underlying POMDP problem would. In POMDPs, an agent observes observation o with probability $\Omega(o|s', a)$. Given a probability distribution b over the state space S, $b(s)$ denotes the probability that the environment is in state s. Given $b(s)$, after taking action a and observing o, the new probability is $b'(s') = \mu\Omega(o|s', a)\sum_{s\in S} T(s'|s, a)b(s)$, where μ is a normalizing constant and $T(s'|s, a)$ is the probability of transitioning from state s to state s' with action a. We prove that the BST augmented with probabilities produces states s at depth k with frequencies proportional to $b^k(s)$, i.e., the k-th POMDP belief state.

Let h be a history with $\beta(h) = s'$, let $c = \{c_1, \ldots, c_n\}$ be the set of impossible nodes after observing o and $\neg\mathsf{c} = \neg\mathsf{c}_1 \cap \cdots \cap \neg\mathsf{c}_n$. In the following we write h_i for $h_{\leq i}$. The search will select h with the probability $P'(\mathsf{h}) = P'_e(h) \cdot P'_e(h_{k-1}) \cdots P'_e(h_0) =$

$$\frac{P(\mathsf{h}|\neg\mathsf{c})}{P(\mathsf{h_{k-1}}|\neg\mathsf{c})} \cdot \frac{P(\mathsf{h_{k-1}}|\neg\mathsf{c})}{P(\mathsf{h_{k-2}}|\neg\mathsf{c})} \cdots \frac{P(\mathsf{h_1}|\neg\mathsf{c})}{P(\mathsf{h_0}|\neg\mathsf{c})} = \frac{P(\mathsf{h}|\neg\mathsf{c})}{P(\mathsf{h_0}|\neg\mathsf{c})},$$

where the denominator equals 1, because the probability of reaching a node below the root is always 1, given a possible leaf is reached. By applying Bayes' rule we get $P'(\mathsf{h}) = P(\mathsf{h}|\neg\mathsf{c}) = \frac{P(\neg\mathsf{c}|\mathsf{h})\cdot P(\mathsf{h})}{P(\neg\mathsf{c})}$, where $P(\neg\mathsf{c}|\mathsf{h}) = 1$, if h is a possible history, since it is a leaf node. So we have $P'(\mathsf{h}) = P(\mathsf{h})/P(\neg\mathsf{c})$. In words, the probability that history h is chosen is the sum of the old probabilities of its possible children, divided by the sum of the old probabilities of all possible histories. Now let us compare that to the POMDP update formula

$$b'(s') = \frac{\Omega(o|s', a)\sum_{s\in S} T(s'|s, a)b(s)}{\sum_{s'\in S}\Omega(o|s', a)\sum_{s\in S} T(s'|s, a)b(s)}.$$

Let us assume the simple case where each history represents exactly

one state. In the numerator, $T(s'|s,a)b(s)$ represents the probability of reaching state s', which is the same as $P(\mathsf{h})$. Ω represents the marking of the node. If $\Omega(o|s',a) = 1$, we will receive observation o in s' with action a and the node is possible. Notice that since every history represents exactly one state, $\Omega(o|s',a)$ is either 1 (which means that we observed o) or 0 (we did not observe o). The same reasoning applies to the denominator, which is just the sum over all possible nodes, since Ω will be 0 for states where observation o is not possible. It follows that $b'(s') = P(\mathsf{h})/P(\neg\mathsf{c}) = P'(\mathsf{h})$, which is what we wanted to show. In the case that there exist multiple h with $\beta(h) = s'$ the same reasoning holds, but $\Omega(o|s',a)$ now lies between 0 and 1, i.e. the proportion of the possible histories representing s'. Additionally, the probability of choosing s' is the sum of the probabilities of choosing histories h with $\beta(h) = s'$.

UCT Search. We still have to show the asymptotic optimality of the UCT algorithm. In the following, we use $\pi : \mathcal{B} \hookrightarrow A$ to refer to a policy of a player, i.e., a mapping from belief states to applicable actions. We reason about the UCT algorithm in several steps:

1. The proportion of visits to state s' when coming from state s and applying action $a = \pi(s)$ is $T(s'|s,a)$, and the proportion of visits of an observation node representing o is $\Omega(o|s',a)$.
2. In the limit, every node will be visited infinitely often.
3. The Q values of action nodes A in the last action-observation layer converge to the expected values of the actions a represented by A.
4. The optimal policy will be followed exponentially more often than all suboptimal policies.
5. The Q values of interior action nodes A converge to the expected values of the actions a represented by A.
6. The UCT algorithm in the AOT tree is asymptotically optimal.

We show the individual claims separately:

1. This follows immediately from the definition of BST search and the traversal of the UCT tree, i.e., BST search provides us with state s with probability according to $b(s)$ and our policy provides us with action a. The UCT algorithm then chooses one random action a_r for the random player and computes the next state $s' = \alpha(s,(a,a_r))$, as well as the next observation $o = \omega(p,s,(a,a_r))$. The proportion of visits of o with s' is then by definition $\Omega(o|s',a) \cdot T(s'|s,a)$.
2. In the limit, every node will be visited infinitely often, which is a general property of UCT search [7].
3. Suppose we are at an action node A in the last action-observation layer. Let $S^A_\star$ be the set of all states contained in any child node of A. Then all $s_\star \in S^A_\star$ must be terminal states. Then it follows from (1) and (2) that each $s_\star \in S^A_\star$ is reached with probability according to $b(s_\star)$ and therefore, $Q(A) = \sum_{s_\star \in S^A_\star} b(s_\star)R(s_\star)$.
4. This follows from the UCT formula.
5. From (3) we know that in the limit the Q values of action nodes in the last action-observation layer are the expected values of the actions they represent. This allows us to inductively compute the Q values of interior action nodes as well. Since (4) holds, it follows that for each interior action node A and each observation node o below A, the successor action $a^*_i(o)$ with maximal $Q(\langle A,o,a^*_i(o)\rangle)$ dominates in the computation of the aggregated value $Q(A)$. Thus, the value of suboptimal actions a_i will not affect $Q(A)$. Therefore,

$$Q(A) = \sum_{\langle A,o\rangle \in \mathcal{O}} Q(\langle A,o,a^*_i(o)\rangle) \sum_{s' \in S'} \Omega(o|s',a) \cdot b(s')$$

which is the expected value of A.
6. This follows from the above using backward induction from leaf nodes to the root of the UCT tree. □

5 EXPERIMENTAL EVALUATION

We evaluated the NORNS algorithm on several GDL-II single-player games, most of them taken from the Australian GGP competition. For each game, we varied the number of simulations per step. One simulation consists of the sampling of one world state s from the current belief state using belief state tree search, followed by one UCT cycle (selection, expansion, simulation, backpropagation) starting in s. After the simulations, the action with the highest action node value is submitted to the game master and a new observation is received. This is repeated until the game reaches a terminal state. We ran each configuration (fixed number of simulations) 100 times and averaged the scores over the 100 runs. Notice that, unlike our formal definition of the reward function R, which returns values in $[0,1]$, in our benchmarks the rewards are natural numbers between 0 and 100.

Mastermind. In *Mastermind*, the random player chooses a pattern of four code pegs, where each peg can take one of four different colors. The other player has to guess a pattern and will observe for which pegs he guessed the correct color. If he finds the correct pattern with at most four guesses, he gets a score of 100. If he needs more than four guesses, the score is reduced by 5 for each additional guess. The left curve in Figure 3 shows the performance of the NORNS algorithm in *Mastermind*. We can see that the average score steadily increases with more simulations, reaching a nearly optimal play after 1000 simulations.

Numberguessing. In *Numberguessing*, the random player chooses a natural number n from a given range, and the other player can either ask whether $n \leq m$ for a number m of his choice from the given range and get the correct answer, or he can decide that he is ready to guess and can report his guess m in the next move. The score depends on the number of questions the player needed to ask. Obviously, binary search is the optimal strategy. If the player needs at most as many steps as binary search would need, he obtains 100 points. Additional steps reduce the score by 30 for the first and by 20 for each further additional step. Guessing the wrong number leads to a score of 0. Asking the same question more than once is inefficient, but legal. This leads to a huge game tree even for small ranges of numbers. We evaluated two versions of the *Numberguessing* game, both with a maximum of 14 possible steps, i.e., the player is forced to guess a number after at most 13 steps. The right curve in Figure 3 shows the evaluation of the NORNS algorithm in the *Numberguessing big* setting, where numbers are chosen from the range from 1 to 16. Like in *Mastermind*, the player performs better with more simulations. But after reaching around 80 points on average, it seems as though more simulations do not affect the score any more. One explanation for this is the huge game tree consisting of around 16^{14} nodes in contrast to the small number of optimal ways to play, corresponding to traces of binary search. Since the first additional step already induces a huge penalty, the average value is heavily affected if the player chooses a suboptimal first move. The third, middle, curve shows the results for a smaller version of *Numberguessing*, with numbers ranging from 1 to 8. The average score quickly reaches roughly 90 points on average, but again, reaching the optimal score would need more simulations than we performed. Notice that even for average rewards around 80 (*Numberguessing big*) and 90 (*Numberguessing small*), the rewards

per run vary widely. For example, among the 100 runs with 10^6 simulations in *Numberguessing big*, 50 gave a score of 100, 27 a score of 70, 19 a score of 50, one a score of 30, and three a score of 0, for an average score of 78.7.

It is worth mentioning that we also compared the results to Fluxii, but since the classic version of Fluxii does not value information gathering moves, the *Numberguessing* results consist of the player always guessing a random number. Unfortunately there does not yet exist a standalone version of the successor to Fluxii, which uses the lifted hyperplay technique [13]. In *Mastermind*, Fluxii scored better with less than 100 simulations and similarly to NORNS for the other configurations. We believe the reason for that is the Hyperplay technique which generates the full belief state and therefore should pull ahead with fewer simulations in small games.

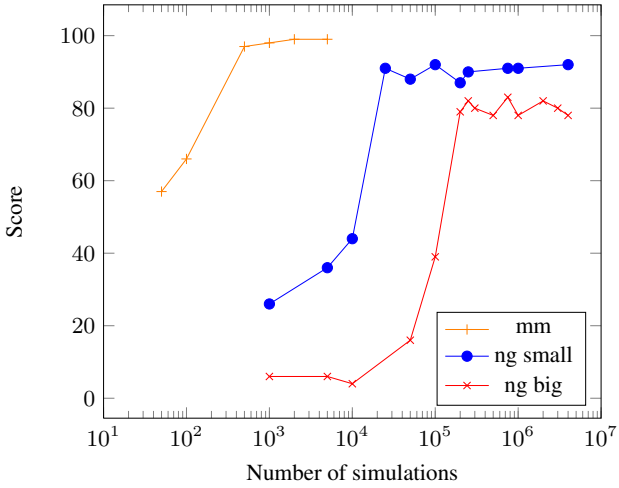

Figure 3: Expected scores as a function of number of simulations. Legend: mm = mastermind, ng = numberguessing.

Smaller Games. We also benchmarked games that are much smaller and only need very few simulations to perform optimally. The first game is the well known *Monty Hall* game, where the show host hides a car behind one of three doors and the player has to choose a door, but is allowed to switch the chosen door after one of the two other doors is opened. The optimal strategy is to always switch the chosen door, and indeed our player always switches the door. The second game is called *Exploding Bomb*. The random player connects one of two possible colored wires with a bomb. Afterwards, the other player can either ask which wire was connected to the bomb, or wait. Asking carries a penalty of 10 points. In the last step, the player has to cut one of the wires, receiving either 100 or 90 points if he defuses the bomb, and 0 if the bomb explodes. Clearly, the best initial action is to ask, with an expected value of 90 points, which is the action our player always performs.

6 CONCLUSION AND FUTURE WORK

We presented an algorithm based on belief state trees and action-observation trees to compute an optimal move in each step of a single-player GDL-II game. By approximation of the belief state, we are able to simulate the past efficiently and can use this information within UCT search for future plays. We avoid the problem of hindsight optimization algorithms by running UCT on action-observation trees. We showed that, given enough time, our algorithm computes optimal actions.

Our main contributions are the definition of the Bayes update within the belief state tree and the use of the action-observation tree for UCT search. Additionally, we gave a proof of asymptotic optimality of the NORNS algorithm and an empirical evaluation.

For future work, we consider it worthwhile investigating how to adapt this single-player algorithm for multi-player games. There are some challenges that come with such games. First, we need some way to model and simulate the plays and strategies of other players. One idea would be to use multiple action-observation trees for UCT, one for each player, which is a similar approach to one also used in classical GGP [4]. We could then use the UCT values to compute initial probabilities in the belief state tree. However, since the beliefs of different players do not have to be the same, we need belief states for other players, which represent our belief about their belief. We still have to find a computationally feasible approach to this problem. However, first experiments, where we use the same belief state for all players, indicate that this approach could pay off.

ACKNOWLEDGEMENTS

This work was partly supported by the DFG as part of the SFB/TR 14 AVACS, the German Aerospace Center (DLR) as part of the Kontiplan project (50 RA 1221) and by BMBF grant 02PJ2667 as part of the KARIS PRO project.

REFERENCES

[1] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer, 'Finite-time Analysis of the Multiarmed Bandit Problem', *Machine Learning*, **47**, 235–256, (2002).
[2] Murray Campbell, A. Joseph Hoane Jr., and Feng hsiung Hsu, 'Deep Blue', *Artificial Intelligence*, **134**(1–2), 57–83, (2002).
[3] Stefan Edelkamp, Tim Federholzner, and Peter Kissmann, 'Searching with Partial Belief States in General Games with Incomplete Information', in *Proc. KI 2012*, pp. 25–36, (2012).
[4] Hilmar Finnsson and Yngvi Björnsson, 'CadiaPlayer: Search Control Techniques', *Künstliche Intelligenz*, **25**(1), 9–16, (2011).
[5] Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. MoGo: A Grid5000-based Software For The Game Of Go. http://www.lri.fr/∼teytaud/mogo.html, 2013.
[6] Michael R. Genesereth, Nathaniel Love, and Barney Pell, 'General Game Playing: Overview of the AAAI Competition', *AI Magazine*, **26**(2), 62–72, (2005).
[7] Levente Kocsis and Csaba Szepesvári, 'Bandit Based Monte-Carlo Planning', in *Proc. ECML 2006*, pp. 282–293, (2006).
[8] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth, 'General Game Playing: Game Description Language Specification', Technical report, Stanford Logic Group, Stanford University, (2008).
[9] Jean Méhat and Tristan Cazenave, 'A Parallel General Game Player', *Künstliche Intelligenz*, **25**(1), 43–47, (2011).
[10] Jonathan Rubin and Ian D. Watson, 'Computer poker: A review', *Artif. Intell.*, **175**(5-6), 958–987, (2011).
[11] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen, 'Checkers Is Solved', *Science*, **317**(5844), 1518–1522, (2007).
[12] Michael Schofield, Timothy Cerexhe, and Michael Thielscher, 'Hyperplay: A Solution to General Game Playing with Imperfect Information', in *Proc. AAAI 2012*, pp. 1606–1612, (2012).
[13] Michael Schofield, Timothy Cerexhe, and Michael Thielscher, 'Lifting HyperPlay for General Game Playing to Incomplete-Information Models', in *Proc. GIGA 2013 Workshop*, pp. 39–45, (2013).
[14] David Silver and Joel Veness, 'Monte-Carlo Planning in Large POMDPs', in *Proc. NIPS 2010*, pp. 2164–2172, (2010).
[15] Michael Thielscher, 'GDL-II', *Künstliche Intelligenz*, **25**(1), 63–66, (2011).