Learning Partial Policies to Speedup MDP Tree Search

Jervis Pinto
School of EECS
Oregon State University
Corvallis OR 97331 USA
pinto@eecs.oregonstate.edu

Alan Fern
School of EECS
Oregon State University
Corvallis OR 97331 USA
afern@eecs.oregonstate.edu

Abstract

A popular approach for online decision making in large MDPs is time-bounded tree search. The effectiveness of tree search, however, is largely influenced by the action branching factor, which limits the search depth given a time bound. An obvious way to reduce action branching is to consider only a subset of potentially good actions at each state as specified by a provided partial policy. In this work, we consider offline learning of such partial policies with the goal of speeding up search without significantly reducing decision-making quality. Our first contribution is to study learning algorithms based on reducing our learning problem to i.i.d. supervised learning. We give a reduction-style analysis of three such algorithms, each making different assumptions, which relates the supervised learning objectives to the sub-optimality of search using the learned partial policies. Our second contribution is to describe concrete implementations of the algorithms within the popular framework of Monte-Carlo tree search. Finally, the third contribution is to evaluate the learning algorithms in two challenging MDPs with large action branching factors, showing that the learned partial policies can significantly improve the anytime performance of Monte-Carlo tree search.

1 INTRODUCTION

Lookahead tree search is a common approach for time-bounded decision making in large Markov Decision Processes (MDPs). Actions are selected by estimating action values at the current state by building a finite-horizon lookahead tree using an MDP model or simulator. A variety of algorithms are available for building such trees, including instances of Monte-Carlo Tree Search (MCTS) such as UCT (Kocsis and Szepesvári, 2006), Sparse Sampling (Kearns et al., 2002), and FSSS (Walsh et al., 2010), along with model-based search approaches such as RTDP (Barto et al., 1995) and AO* (Bonet and Geffner, 2012). Given time constraints, however, the performance of these approaches depends on the action branching factor, which is often considerable and greatly limits the feasible search depth. An obvious way to address this problem is to provide prior knowledge for explicitly pruning bad actions from consideration. In this paper, we consider offline learning of such prior knowledge in the form of partial policies.

A partial policy is a function that quickly returns an action subset for each state and can be integrated into search by pruning away actions not included in the subsets. Thus, a partial policy can significantly speedup search if it returns small action subsets, provided that the overhead for applying the partial policy is small enough. If those subsets typically include high-quality actions, then we might expect little decrease in decision-making quality. Although learning partial policies to guide tree search is a natural idea, it has received surprisingly little attention, both in theory and practice. In this paper we formalize this learning problem from the perspective of "speedup learning". We are provided with a distribution over search problems in the form of a root state distribution and a search depth bound. The goal is to learn partial policies that significantly speedup depth D search, while bounding the expected regret of selecting actions using the pruned search versus no pruning.

In order to solve this learning problem, there are at least two key choices that must be made: 1) Selecting a training distribution over states arising in lookahead trees, and 2) Selecting a loss function that the partial policy is trained to minimize with respect to the chosen distribution. The key contribution of our work is to consider a family of reduction-style algorithms that answer these questions in different ways. In particular, we consider three algorithms that reduce partial policy learning to i.i.d. supervised learning problems characterized by choices for (1) and (2). Our main results bound the sub-optimality of tree search using the learned partial policies in terms of the expected loss of the supervised learning problems. Interestingly, these results for learning partial policies mirror similar reduction-style results for learning (complete) policies via imitation, e.g. (Ross and Bagnell, 2010; Syed and Schapire, 2010; Ross et al., 2011).

We empirically evaluate our algorithms in the context of learning partial policies to speedup MCTS in two challenging domains with large action branching factors: 1) the classic dice game, Yahtzee, and 2) a real-time strategy game, Galcon. The results show that using the learned partial policies to guide MCTS leads to significantly improved anytime performance in both domains. Furthermore, we show that several other existing approaches for injecting knowledge into MCTS are not as effective as using partial policies for action pruning and can often hurt search performance rather than help.

2 PROBLEM SETUP

We consider sequential decision making in the framework of Markov Decision Processes (MDPs) and assume basic familiarity. An MDP is a tuple (S, A, P, R), where S is a finite set of states, A a finite set of actions, $P(s'|s, a)$ is the transition probability of arriving at state s' after executing action a in state s, and $R(s, a) \in [0, 1]$ is the reward function giving the reward of taking action a in state s. The typical goal of MDP planning and learning is to compute a policy for selecting an action in any state, such that following the policy (approximately) maximizes some measure of long-term expected reward.

In practice, regardless of the long-term reward measure, for large MDPs, the offline computation of high-quality policies over all environment states is impractical. In such cases, a popular action-selection approach is online tree search, where at each encountered environment state, a time-bounded search is carried out in order to estimate action values. Note that this approach requires the availability of either an MDP model or an MDP simulator in order to construct search trees. In this paper, we assume that a model or simulator is available and that online tree search has been chosen as the action selection mechanism. Next, we formally describe the paradigm of online tree search, introduce the notion of partial policies for pruning tree search, and then formulate the problem of offline learning of such partial policies.

Online Tree Search. We will focus on depth-bounded search assuming a search depth bound of D throughout, which bounds the length of future action sequences to be considered. Given a state s, we denote by $T(s)$ the depth D expectimax tree rooted at s. $T(s)$ alternates between layers of state nodes and action nodes, labeled by states and actions respectively. The children of each state node are action nodes for each action in A. The children of an action node a with parent labeled by s are all states s' such that $P(s'|s, a) > 0$. Figure 1(a) shows an example of a depth two expectimax tree. The depth of a state node is the number of action nodes traversed from the root to reach it, noting that leaves of $T(s)$ will always be action nodes.

The optimal value of a state node s at depth d, denoted by $V_d^*(s)$, is equal to the maximum value of its child action nodes, which we denote by $Q_d^*(s, a)$ for child a. We define $Q_d^*(s, a)$ to be $R(s, a)$ if $d = D - 1$ (i.e. for leaf action nodes) and otherwise $R(s, a) + E\left[V_{d+1}^*(s')\right]$, where $s' \sim P(\cdot|s, a)$ ranges over children of a. The optimal action policy for state s at depth d will be denoted by $\pi_d^*(s) = \arg\max_a Q_d^*(s, a)$. Given an environment state s, online search algorithms such as UCT or RTDP attempt to completely or partially search $T(s)$ in order to approximate the root action values $Q_0^*(s, a)$ well enough to identify the optimal action $\pi_d^*(s)$. It is important to note that optimality in our context is with respect to the specified search depth D, which may be significantly smaller than the expected planning horizon in the actual environment. This is a practical necessity that is often referred to as receding-horizon control. Here we simply assume that an appropriate search depth D has been specified and our goal is to speedup planning within that depth.

Search with Partial Policies. One way to speedup depth D search is to prune actions from $T(s)$. In particular, if a fixed fraction σ of actions are removed from each state node, then the size of the tree would decrease by a factor of $(1-\sigma)^D$, potentially resulting in significant computational savings. For this purpose we will utilize partial policies. A depth D (non-stationary) partial policy ψ is a sequence $(\psi_0, \ldots, \psi_{D-1})$ where each ψ_d maps a state to an action subset. Given a partial policy ψ and root state s, we can define a pruned tree $T_\psi(s)$ that is identical to $T(s)$, except that each state s at depth d only has subtrees for actions in $\psi_d(s)$, pruning away subtrees for any child $a \notin \psi_d(s)$. Figure 1(b) shows a pruned tree $T_\psi(s)$, where ψ prunes away one action at each state. It is straightforward to incorporate ψ into a search algorithm by only expanding actions at state nodes that are consistent with ψ.

We define the state and action values relative to $T_\psi(s)$ the same as above and let $V_d^\psi(s)$ and $Q_d^\psi(s, a)$ denote the depth d state and action value functions. We will denote the highest-valued, or greedy, root action of $T_\psi(s)$ by $G_\psi(s) = \arg\max_{a \in \psi_0(s)} Q_0^\psi(s, a)$. This is the root action that a depth D search procedure would attempt to return in the context of ψ. Note that a special case of partial policies is when $|\psi_d(s)| = 1$ for all s and d, which means that ψ defines a traditional (complete) deterministic MDP policy. In this case, V_d^ψ and Q_d^ψ represent the traditional depth d state value and action value functions for policies. We say that a partial policy ψ subsumes a partial policy ψ' if for each s and d, $\psi'_d(s) \subset \psi_d(s)$. In this case, it is straightforward to show that for any s and d, $V_d^{\psi'}(s) \leq V_d^\psi(s)$.

Clearly, a partial policy can reduce the complexity of search. However, we are also concerned with the quality of decision making using $T_\psi(s)$ versus $T(s)$, which we will quantify in terms of expected regret. The regret of selecting action a at state s relative to $T(s)$ is equal to $V_0^*(s) - Q_0^*(s, a)$, noting that the regret of the optimal action $\pi_0^*(s)$ is zero. We prefer partial policies that result in root decisions with small regret over the root state distribution that we expect to encounter, while also supporting significant pruning. For this purpose, if μ_0 is a distribution over root states, we define the expected regret of ψ with respect to μ_0 as $\text{REG}(\mu_0, \psi) = E\left[V_0^*(s_0) - Q_0^*(s_0, G_\psi(s_0))\right]$, where $s_0 \sim \mu_0$.

Learning Problem. We consider an offline learning setting where we are provided with a model or simulator of the MDP in order to train a partial policy that will be used later for online decision making. That is, the learning problem provides us with a distribution μ_0 over root states (or a representative sample from μ_0) and a depth bound D. The intention is for μ_0 to be representative of the states that will be encountered during online use. In this work, we are agnostic about how μ_0 is defined for an application. However, a typical example and one used in our experiments, is for μ_0 to correspond to the distribution of states encountered along trajectories of a receding horizon controller that makes decisions based on unpruned depth D search.

Given μ_0 and D, our "speedup learning" goal is to learn a partial policy ψ with small expected regret $\text{REG}(\mu_0, \psi)$, while providing significant pruning. That is, we want to approximately imitate the decisions of depth D unpruned search via a much less expensive depth D pruned search. In general, there will be a tradeoff between the potential speedup and expected regret. At one extreme, it is always possible to achieve zero expected regret by selecting a partial policy that does no pruning and hence no speedup. At the other extreme, we can remove the need for any search by learning a partial policy that always returns a single action (i.e. a complete policy). However, for many complex MDPs, it can be difficult to learn computationally efficient, or reactive, policies that achieve small regret. Rather, it may be much easier to learn partial policies that prune away many, but not all actions, yet still retain high-quality actions. While such partial policies lead to more search than a reactive policy, the regret can be much less.

In practice, we seek a good tradeoff between the two extremes, which will depend on the application. Instead of specifying a particular trade-off point as our learning objective, we develop learning algorithms in the next section that provide some ability to explore different points. In particular, the algorithms are associated with regret bounds in terms of supervised learning objectives that measurably vary with different amounts of pruning.

Algorithm 1 A template for learning a partial policy $\psi = (\psi_0, \ldots, \psi_{D-1})$. The template is instantiated by specifying the pairs of distributions and cost functions (μ_d, C_d) for $d \in \{0, \ldots, D-1\}$. LEARN is an i.i.d. supervised learning algorithm that aims to minimize the expected cost of each ψ_d relative to C_d and μ_d.

```
procedure PARTIALPOLICYLEARNER($\{(\mu_d, C_d)\}$)
    for $d = 0, 1, \ldots, D-1$ do
        Sample a training set of states $S_d$ from $\mu_d$
        $\psi_d \leftarrow$ LEARN($S_d, C_d$)
    end for
    return $\psi = (\psi_0, \psi_1, \ldots, \psi_{D-1})$
end procedure
```

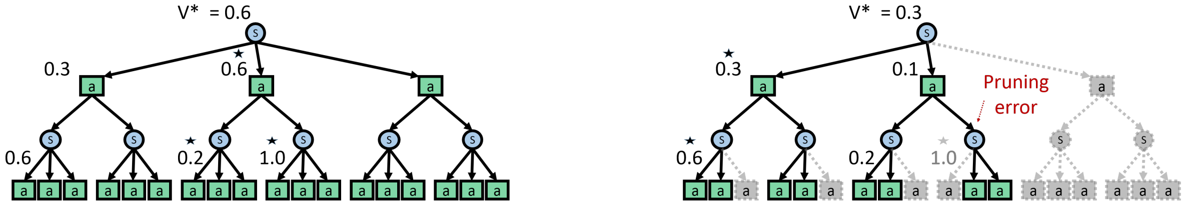
3 LEARNING PARTIAL POLICIES

Given μ_0 and D, we now develop reduction-style algorithms for learning partial policies. The algorithms reduce partial policy learning to a sequence of D i.i.d. supervised learning problems, each producing one partial policy component ψ_d. The supervised learning problem for ψ_d will be characterized by a pair (μ_d, C_d), where μ_d is a distribution over states, and C_d is a cost function that, for any state s and action subset $A' \subseteq A$ assigns a prediction cost $C_d(s, A')$. The cost function is intended to measure the quality of A' with respect to including actions that are high quality for s. Typically the cost function is monotonically decreasing in A' and $C_d(s, A) = 0$.

In this section we assume the availability of an i.i.d. supervised learner LEARN that takes as input a set of states drawn from μ_d, along with C_d, and returns a partial policy component ψ_d that is intended to minimize the expected cost of ψ_d on μ_d, i.e. minimize $E\left[C_d(s, \psi_d(s))\right]$ for $s \sim \mu_d$. In practice, the specific learner will depend on the cost function and we describe our particular implementations in Section 4. Rather, in this section, we focus on defining reductions that allow us to bound the expected regret of ψ by the expected costs of the ψ_d returned by LEARN. The generic template for our reduction algorithms is shown in Algorithm 1.

In the following, our state distributions will be specified in terms of distributions induced by (complete) policies. In particular, given a policy π, we let $\mu_d(\pi)$ denote the state distribution induced by drawing an initial state from μ_0 and then executing π for d steps. Since we have assumed an MDP model or simulator, it is straightforward to sample from $\mu_d(\pi)$ for any provided π. Before proceeding we state a simple proposition that will be used to prove our regret bounds.

Proposition 3.1. *If a complete policy π is subsumed by partial policy ψ, then for any initial state distribution μ_0, REG$(\mu_0, \psi) \leq E\left[V_0^*(s_0)\right] - E\left[V_0^\pi(s_0)\right]$, for $s_0 \sim \mu_0$.*



(a) Unpruned expectimax tree $T(s)$ with $D = 2$

(b) Pruning T with **partial** policy ψ gives T_ψ where a third of the actions have been pruned.

Figure 1: Unpruned and pruned expectimax trees with depth $D = 2$ for an MDP with $|A| = 3$ and two possible next states.

Proof. Since π is subsumed by ψ, we know that $Q_0^\psi(s, G_\psi(s)) = V_0^\psi(s) \geq V_0^\pi(s)$. Since for any a, $Q_0^*(s,a) \geq Q_0^\psi(s,a)$, we have for any state s, $Q_0^*(s, G_\psi(s)) \geq V_0^\pi(s)$. The result follows by negating each side of the inequality, followed by adding $V_0^*(s)$, and taking expectations. □

Thus, we can bound the regret of a learned ψ if we can guarantee that it subsumes a policy whose expected value has bounded sub-optimality. Next we present three reductions for doing this, each having different requirements and assumptions.

3.1 OPI : OPTIMAL PARTIAL IMITATION

Perhaps the most straightforward idea for learning a partial policy is to attempt to find a partial policy that is usually consistent with trajectories of the optimal policy π^*. That is, each ψ_d should be learned so as to maximize the probability of containing actions selected by π_d^* with respect to the optimal state distribution $\mu_d(\pi^*)$. This approach is followed by our first algorithm called Optimal Partial Imitation (OPI). In particular, Algorithm 1 is instantiated with $\mu_d = \mu_d(\pi^*)$ (noting that $\mu_0(\pi^*)$ is equal to μ_0 as specified by the learning problem) and C_d equal to zero-one cost. Here $C_d(s, A') = 0$ if $\pi_d^*(s) \in A'$ and $C_d(s, A') = 1$ otherwise. Note that the expected cost of ψ_d in this case is equal to the probability that ψ_d does not contain the optimal action, which we will denote by $e_d^*(\psi) = \Pr_{s \sim \mu_d(\pi^*)}(\pi_d^*(s) \notin \psi_d(s))$.

A naive implementation of OPI is straightforward. We can generate length D trajectories by drawing an initial state from μ_0 and then selecting actions (approximately) according to π_d^* using standard unpruned search. Defined like this, OPI has the nice property that it only requires the ability to reliably compute actions of π_d^*, rather than requiring that we also estimate action values accurately. This allows us to exploit the fact that search algorithms such as UCT often quickly identify optimal actions, or sets of near-optimal actions, well before the action values have converged.

Intuitively we should expect that if the expected cost e_d^* is small for all d, then the regret of ψ should be bounded, since the pruned search trees will generally contain optimal actions for state nodes. The following clarifies this dependence. For the proof, given a partial policy ψ, it is useful to define a corresponding complete policy ψ^+ such that $\psi_d^+(s) = \pi_d^*(s)$ whenever $\pi_d^*(s) \in \psi_d(s)$ and otherwise $\psi_d^+(s)$ is the lexicographically least action in $\psi_d(s)$. Note that ψ^+ is subsumed by ψ.

Theorem 3.2. *For any initial state distribution μ_0 and partial policy ψ, if for each $d \in \{0, \ldots, D-1\}$, $e_d^*(\psi) \leq \epsilon$, then REG$(\mu_0, \psi) \leq \epsilon D^2$.*

Proof. Given the assumption that $e_d^*(\psi) \leq \epsilon$ and that ψ^+ selects the optimal action whenever ψ contains it, we know that $e_d^*(\psi^+) \leq \epsilon$ for each $d \in \{0, \ldots, D\}$. Given this constraint on ψ^+ we can apply Lemma 3[1] from (Syed and Schapire, 2010) which implies $E[V_0^{\pi^+}(s_0)] \geq E[V_0^*(s_0)] - \epsilon D^2$, where $s_0 \sim \mu_0$. The result follows by combining this with Prop. 3.1. □

This result mirrors work on reducing imitation learning to supervised classification (Ross and Bagnell, 2010; Syed and Schapire, 2010), showing the same dependence on the planning horizon. While space precludes details, it is straightforward to construct an example problem where the above regret bound is shown to be tight. This result motivates a learning approach where we have LEARN attempt to return ψ_d that each maximizes pruning (returns small action sets) while maintaining a small expected cost.

3.2 FT-OPI : FORWARD TRAINING OPI

OPI has a potential weakness, similar in nature to issues identified in prior work on imitation learning (Ross and Bagnell, 2010; Ross et al., 2011). In short, OPI does not train ψ to recover from its own pruning mistakes. Consider a node n in the optimal subtree of a tree $T(s_0)$ and suppose

[1] The main result of (Syed and Schapire, 2010) holds for stochastic policies and requires a more complicated analysis that results in a looser bound. Lemma 3 is strong enough for deterministic policies.

that the learned ψ erroneously prunes the optimal child action of n. This means that the optimal subtree under n will be pruned from $T_\psi(s)$, increasing the potential regret. Ideally, we would like the pruned search in $T_\psi(s)$ to recover from the error gracefully and return an answer based on the best remaining subtree under n. Unfortunately, the distribution used to train ψ by OPI was not necessarily representative of this alternate subtree under n, since it was not an optimal subtree of $T(s)$. Thus, no guarantees about the pruning accuracy of ψ can be made under n.

In imitation learning, this type of problem has been dealt with via "forward training" of non-stationary policies (Ross et al., 2011) and a similar idea works for us. The Forward Training OPI (FT-OPI) algorithm differs from OPI only in the state distributions used for training. The key idea is to learn the partial policy components ψ_d in sequential order from $d = 0$ to $d = D - 1$ and then training ψ_d on a distribution induced by $\psi_{0:d-1} = (\psi_0, \ldots, \psi_{d-1})$, which will account for pruning errors made by $\psi_{0:d-1}$. Specifically, recall that for a partial policy ψ, we defined ψ^+ to be a complete policy that selects the optimal action if it is consistent with ψ and otherwise the lexicographically least action. The state distributions used to instantiate FT-OPI is $\mu_d = \mu_d(\psi^+_{0:d-1})$ and the cost function remains zero-one cost as for OPI. We will denote the expected cost of ψ_d in this case by $e^+_d(\psi) = \Pr_{s\sim\mu_d(\psi^+_{0:d-1})}(\pi^*_d(s) \notin \psi_d(s))$, which gives the probability of pruning the optimal action with respect to the state distribution of $\psi^+_{0:d-1}$.

Note that as for OPI, we only require the ability to compute π^* in order to sample from $\mu_d(\psi^+_{0:d-1})$. In particular, note that when learning ψ_d, we have $\psi_{0:d-1}$ available. Hence, we can sample a state for μ_d by executing a trajectory of $\psi^+_{0:d-1}$. Actions for ψ^+_d can be selected by first computing π^*_d and selecting it if it is in ψ_d and otherwise selecting the lexicographically least action.

As shown for the forward training algorithm for imitation learning (Ross et al., 2011), we give below an improved regret bound for FT-OPI under an assumption on the maximum sub-optimality of any action. The intuition is that if it is possible to discover high-quality subtrees, even under sub-optimal action choices, then FT-OPI can learn on those trees and recover from errors.

Theorem 3.3. *Assume that for any state s, depth d, and action a, we have $V^*_d(s) - Q^*_d(s,a) \leq \Delta$. For any initial state distribution μ_0 and partial policy ψ, if for each $d \in \{0, \ldots, D-1\}$, $e^+_d(\psi) \leq \epsilon$, then $REG(\mu_0, \psi) \leq \epsilon\Delta D$.*

Proof. The theorem's assumptions imply that ψ^+ and the search tree $T(s)$ satisfies the conditions of Theorem 2.2 of (Ross et al., 2011). Thus we can infer that $E[V^{\pi^+}_0(s_0)] \geq E[V^*_0(s_0)] - \epsilon\Delta D$, where $s_0 \sim \mu_0$. The result follows by combining this with Proposition 3.1. □

Thus, when Δ is significantly smaller than D, FT-OPI has the potential to outperform OPI given the same bound on zero-one cost. In the worst case $\Delta = D$ and the bound will equal to that of OPI.

3.3 FT-QCM: FORWARD TRAINING Q-COST MINIMIZATION

While FT-OPI addressed one potential problem with OPI, they are both based on zero-one cost, which raises other potential issues. The primary weakness of using zero-one cost is its inability to distinguish between highly sub-optimal and slightly sub-optimal pruning mistakes. It was for this reason that FT-OPI required the assumption that all action values had sub-optimality bounded by Δ. However, in many problems, including those in our experiments, that assumption is unrealistic, since there can be many highly sub-optimal actions (e.g. ones that result in losing a game). This motivates using a cost function that is sensitive to the sub-optimality of pruning decisions.

In addition, it can often be difficult to learn a ψ that achieves both, a small zero-one cost and also provides significant pruning. For example, in many domains, in some states there will often be many near-optimal actions that are difficult to distinguish from the slightly better optimal action. In such cases, achieving low zero-one cost may require producing large action sets. However, learning a ψ that provides significant pruning while reliably retaining at least one near-optimal action may be easily accomplished. This again motivates using a cost function that is sensitive to the sub-optimality of pruning decisions, which is accomplished via our third algorithm, Forward Training Q-Cost Minimization (FT-QCM)

The cost function of FT-QCM is the minimum sub-optimality, or Q-cost, over unpruned actions. In particular, we use $C_d(s, A') = V^*_d(s) - \max_{a\in A'} Q^*_d(s,a)$. Our state distribution will be defined similarly to that of FT-OPI, only we will use a different reference policy. Given a partial policy ψ, define a new complete policy $\psi^* = (\psi^*_0, \ldots, \psi^*_{D-1})$ where $\psi^*_d(s) = \arg\max_{a\in\psi_d(s)} Q^*_d(s,a)$, so that ψ^* always selects the best unpruned action. We define the state distributions for FT-QCM as the state distribution induced by ψ^*, i.e. $\mu_d = \mu_d(\psi^*_{0:d-1})$. We will denote the expected Q-cost of ψ at depth d to be $\Delta_d(\psi) = E\left[V^*_d(s_d) - \max_{a\in\psi_d(s_d)} Q^*_d(s_d,a)\right]$, where $s_d \sim \mu_d(\psi^*_{0:d-1})$.

Unlike OPI and FT-OPI, this algorithm requires the ability to estimate action values of sub-optimal actions in order to sample from μ_d. That is, sampling from μ_d requires generating trajectories of ψ^*_d, which means we must be able to accurately detect the action in $\psi_d(s)$ that has maximum value, even if it is a sub-optimal action. The additional overhead for doing this during training depends on the search algorithm being used. For many algorithms, near-optimal actions will tend to receive more attention than clearly sub-optimal actions. In those cases, as long as $\psi_d(s)$ includes reasonably good actions, there may be little additional regret. The following regret bound motivates the FT-QCM algorithm.

Theorem 3.4. *For any initial state distribution μ_0 and partial policy ψ, if for each $d \in \{0, \ldots, D-1\}$, $\Delta_d(\psi) \leq \Delta$, then REG$(\mu_0, \psi) \leq 2D^{\frac{3}{2}}\sqrt{\Delta}$.*

Proof. Consider any pair of non-negative real numbers (ϵ, δ) such that for any d, the probability is no more than ϵ of drawing a state from $\mu_d(\psi^*)$ with Q-cost (wrt ψ) greater than δ. That is $\Pr(C_d(s_d, \psi_d(s_d)) \geq \delta) \leq \epsilon$. We will first bound REG(μ_0, ψ) in terms of ϵ and δ.

Let Π_δ be the set of policies for which all selected actions have regret bounded by δ. It can be shown by induction on the depth that for any $\pi \in \Pi_\delta$ and any state s, $V_0^\pi(s) \geq V_0^*(s) - \delta D$. For the chosen pair (ϵ, δ) we have that for a random trajectory t of ψ^* starting in a state drawn from μ_0 there is at most an ϵD probability that the Q-cost of ψ^* on any state of t is greater than δ. Thus, compared to a policy that always has Q-cost bounded by δ, the expected reduction in total reward of ψ for initial state distribution μ_0 is no more than ϵD^2. This shows that

$$\begin{aligned} E\left[V_0^{\psi^*}(s_0)\right] &\geq \min_{\pi \in \Pi_\delta} E\left[V_0^\pi(s_0)\right] - \epsilon D^2 \\ &\geq E\left[V_0^*(s_0)\right] - \delta D - \epsilon D^2 \end{aligned}$$

Since ψ^* is subsumed by ψ, Proposition 3.1 implies that REG$(\mu_0, \psi) \leq \delta D + \epsilon D^2$.

We now reformulate the above bound in terms of the bound on expected Q-cost Δ assumed by the theorem. Since Q-costs are non-negative, we can apply the Markov inequality to conclude that $\Pr(C_d(s_d, \psi_d(s_d)) \geq \delta) \leq \frac{\Delta_d(\psi)}{\delta} \leq \frac{\Delta}{\delta}$. The pair $(\frac{\Delta}{\delta}, \delta)$ then satisfies the above condition for ψ^*. Thus, we get REG$(\mu_0, \psi) \leq \delta D + \frac{\Delta}{\delta} D^2$. The bound is minimized at $\delta = \sqrt{D\Delta}$, which yields the result. □

FT-QCM tries to minimize this regret bound by minimizing $\Delta_d(\psi)$ via supervised learning at each step. Importantly, as we will show in our experiments, it is often possible to maintain small expected Q-cost with significant pruning, while the same amount of pruning would result in a much larger zero-one cost. It is an open problem as to whether this bound is tight in the worst case.

4 IMPLEMENTATION DETAILS

In this section, we describe our concrete implementation of the above abstract algorithms using the MCTS algorithm UCT (Kocsis and Szepesvári, 2006) for tree search.

UCT is a popular MCTS algorithm, perhaps most famous for leading to significant advances in computer Go (Gelly et al., 2006; Gelly and Silver, 2007). Space precludes a complete description and we only outline the high-level ideas and issues relevant to our work. Given a root state and a time budget, UCT executes a series of Monte-Carlo simulations in order to build an asymmetric search tree that explores more promising parts of the tree first, compared to less promising parts. The key idea behind UCT is to utilize a variant of the multi-armed bandit algorithm UCB (Auer et al., 2002) for selecting actions during the Monte-Carlo simulations. The objective is to balance the exploration of less frequently visited parts of the tree versus exploring nodes that look most promising. State and action values for nodes in the UCT tree are estimated based on the average reward of Monte-Carlo trajectories that go through the nodes. In the limit, UCT will converge to optimal state and action values. In practice, the values computed in a UCT tree after a finite time budget are most accurate for higher quality actions, since they tend to be explored more often. It is straightforward to integrate a partial policy into UCT, by simply removing pruned actions from consideration.

Partial Policy Representation. Our partial policies operate by simply ranking the actions at a state and then pruning a percentage of the bottom actions. Specifically, partial policy components have the form $\psi_d(s \mid w_d, \sigma_d)$, parameterized by an n-dimensional weight vector w_d and pruning fraction $\sigma_d \in [0, 1)$. Given a state s, each action a is ranked according to the linear ranking function $f_d(s, a) = w_d^T \phi(s, a)$, where $\phi(s, a)$ is a user provided n-dimensional feature vector that describes salient properties of state-action pairs. Using f_d we can define a total order over actions, breaking ties lexicographically. $\psi_d(s \mid w_d, \sigma_d)$ is then equal to the set of the $\lceil (1 - \sigma_d)|A| \rceil$ highest ranked actions. This representation is useful as it allows us to separate the training of f_d from the selection of the pruning fraction.

Generating Training States. Each of our algorithms requires sampling training states from trajectories of particular policies that either require approximately computing π_d^* (OPI and FT-OPI) or also action values for sub-optimal actions (FT-QCM). Our implementation of this is to first generate a set of trees using substantial search, which provides us with the required policy or action values and then to sample trajectories from those trees.

More specifically, our learning algorithm is provided with a set of root states S_0 sampled from the target root distribution μ_0. For each $s_0 \in S_0$ we run UCT for a specified time bound, which is intended to be significantly longer than the eventual online time bound. Note that the resulting trees will typically have a large number of nodes on the tree fringe that have been explored very little and hence will not be useful for learning. Because of this we select a depth bound D for training such that nodes at that depth or less have been sufficiently explored and have meaningful action values. The trees are then pruned until depth D.

Given this set of trees we can then generate trajectories using the MDP simulator of the policies specified for each algorithm. For example, OPI simply requires running trajectories through the trees based on selecting actions according to the optimal action estimates. The state at depth d along each trajectory is added to the data set for training ψ_d. FT-QCM samples states for training ψ_d by generating length d trajectories of $\psi^*_{0:d-1}$. Each such action selection requires referring to the estimated action values and returning the best one that is not pruned. The final state on the trajectory is then added to the training set for ψ_d. Note that since our approach generates multiple trajectories from each tree the training sets for each ψ_d are not truly i.i.d. This is an acceptable trade-off in practice since tree construction is expensive and this approach allows for many examples to be generated per tree.

Supervised Learner. It remains to specify how we implement the LEARN procedure. Each training set will consist of pairs $\{(s_i, c_i)\}$ where s_i is a state and c_i is a vector that assigns a cost to each action. For OPI and FT-OPI the cost vector assigns 0 to the optimal action and 1 to all other actions. For FT-QCM the c_i give the Q-costs of each action. We learn the partial policy by first learning the ranking function in a way that attempts to rank low-cost actions as highly as possible and then select an appropriate pruning percentage based on the learned ranker.

For rank learning, we follow a common approach of converting the problem to cost-sensitive binary classification. In particular, for a given example (s, c) we create a cost-sensitive classification example for each pair of actions a_1 and a_2 of the form $(\phi(s, a_1) - \phi(s, a_2), c(a_2) - c(a_1))$. Learning a linear classifier for such an example will attempt to rank a_1 above or below a_2 according to the cost difference. We apply an existing cost-sensitive learner (Langford, 2011) to learn a weight vector based on the pairwise data. Note that for zero-one loss, we do not create pairwise examples involving just sub-optimal actions since their cost difference will always be zero.

Finally, after learning the ranking function for a particular ψ_d, we must select an appropriate pruning percentage σ_d. In practice, we do this by analyzing the expected cost of ψ_d for a range of pruning values and select a pruning value that yields reasonably small costs. Section 6 gives details of the selections for our experiments.

5 RELATED WORK

While there is a large body of work on integrating learning and planning, to the best of our knowledge, we do not know of any work on learning partial policies for speeding up online MDP planning.

There are a number of efforts that study model-based reinforcement learning (RL) for large MDPs that utilize tree search methods for planning with the learned model, e.g. RL using FSSS (Walsh et al., 2010), Monte-Carlo AIXI (Veness et al., 2011), and TEXPLORE (Hester and Stone, 2013). However, these methods focus on model/simulator learning and do not attempt to learn to speedup tree search using the learned models, which is the focus of our work.

A more related body of work is on learning search control knowledge in deterministic planning and games. One thread of work has been on learning knowledge for STRIPS-style deterministic planners, e.g. learning heuristics and policies for guiding best-first search (Yoon et al., 2008) or state ranking functions (Xu et al., 2009). The problem of learning improved leaf evaluation heuristics has also been studied in the context of deterministic real-time heuristic search (Bulitko and Lee, 2006). As one more example, evaluation functions have been learned for game tree search based on learning from “principle variations” of deep searches (Veness et al., 2009). The training data for this approach is similar in spirit to that of our OPI algorithm. Since these algorithms have been developed for deterministic settings, it is not straightforward to adapt them to the general MDP setting. Further, none of these existing methods, to our knowledge, have provided a theoretical analysis of the possible regret of using the learned knowledge, which is one of our main contributions.

There have been a number of efforts for utilizing domain-specific knowledge in order to improve/speedup MCTS, many of which are covered in a recent survey (Browne et al., 2012). A popular technique is to use a bias term $f(s, a)$ for guiding action selection during search. f is hand-provided in (Chaslot et al., 2007; Couëtoux et al., 2011) and learned in (Gelly and Silver, 2007; Sorg et al., 2011). Generally there are a number of parameters that dictate how strongly $f(s, a)$ influences search and how that influence decays as search progresses. In our experience, tuning the parameters for a particular problem can be tedious and difficult to do in a domain-independent way. Similar issues hold for the approach in (Gelly and Silver, 2007) which attempts to learn an approximation of Q^* and then initializes search nodes with the estimate. In (Sorg et al., 2011), control knowledge is learned via policy-gradient techniques in the form of a reward function and used to guide MCTS with the intention of better performance given a time budget. So far, however, the approach has not been analyzed formally and has not been demonstrated on large MDPs. Experiments in small MDPs have also not demonstrated improvement in terms of wall clock time over vanilla MCTS.

Finally, MCTS methods often utilize hand-coded or learned “default policies” (e.g., MoGo (Gelly et al., 2006)) to improve anytime performance. While this has shown some promise in specific domains such as Go, where the policies can be highly engineered for efficiency, we have found that the computational overhead of using learned default policies is often too high a price to pay. In particular, most such learned policies require the evaluation of a feature vector at each state encountered, or for each state-action pair. This may cause orders of magnitude fewer trajectories to be executed compared to vanilla MCTS. In our experience, this can easily lead to degraded performance per unit time. Furthermore, there is little formal understanding about how to learn such rollout policies in principled ways, with straightforward approaches often yielding decreased performance (Silver and Tesauro, 2009).

6 EXPERIMENTS

We consider learning partial policies in two domains.

Yahtzee is a classic dice game with ≈ 17 actions on average. Actions correspond to selecting subsets of dice to roll and selecting categories to score. The objective is to maximize the sum of category scores. We use 26 features for learning the partial policy which encode the impact of a roll action or select action for each of the 13 categories.

Galcon is a two-player real-time strategy game with approximately 174 actions per move on average (max. ≈ 800). The objective is to maximize one's own population on a 2-D grid of planets by directing variable-sized fleets of units between planets to capture enemy planets or fortify one's own. Transitions are stochastic when opposing populations battle over a planet. We use 20 real-valued state-action features that coarsely encode the impact of an action on population size, planet ownership, etc.

In what follows, we experiment with the two extreme algorithms, OPI and FT-QCM. The intermediate algorithm FT-OPI is excluded in this paper due to the time required to perform large-scale evaluations across a wide spectrum of time bounds. Preliminary experiments suggest that FT-OPI performs between OPI and FT-QCM.

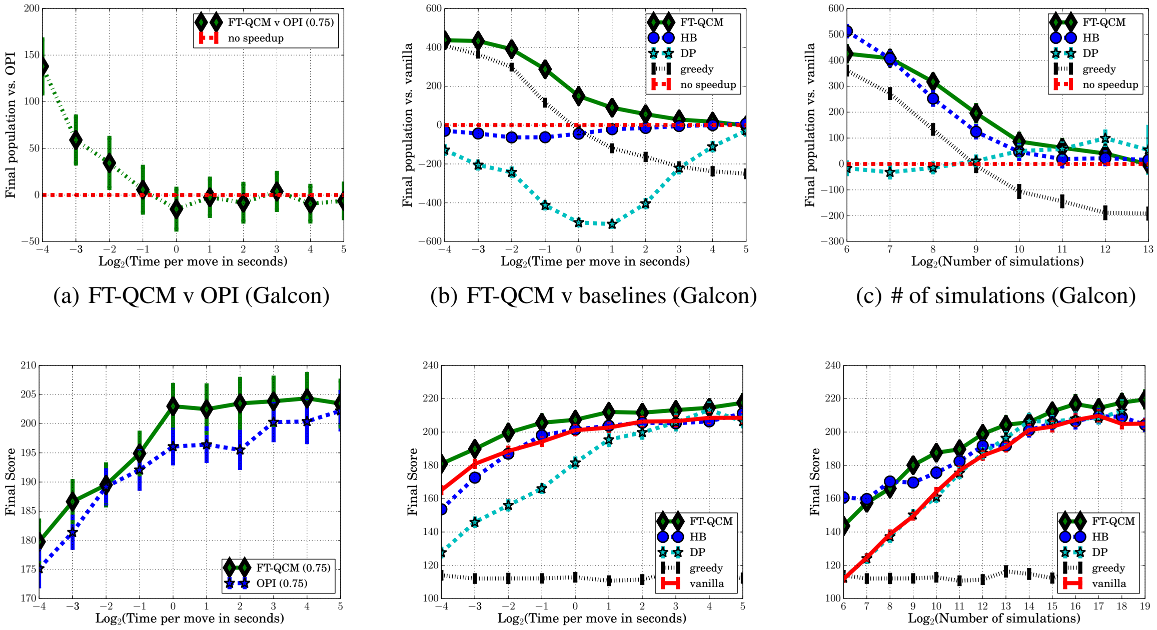
Setup. For training we provide our learning algorithms with root states generated by playing 100 full games allowing UCT 64 seconds per move per tree. This produces thousands of root states along with the search trees, which we use for training as described in Section 4. Due to the large action and stochastic branching factors, even with 64 seconds per move, we selected $D = 3$ for learning since the value estimates produced by UCT for deeper nodes were not accurate. Note that in our evaluations we do not limit the rollouts of UCT to depth 3, rather the rollouts proceed until terminal states, which provide a longer term heuristic evaluation for tree nodes.[2] In order to select the pruning fractions σ_d, in results not shown for space reasons, we plotted the supervised losses achieved for various learned ranking functions for a variety of σ_d. We found that in Galcon a value of $\sigma_d = 0.75$ provided a good trade-off point since the supervised average costs began to sharply increase beyond that point. For Yahtzee we found that $\sigma_d = 0.75$ was a good trade-off point for all depths except the root. At the root we used a less aggressive pruning fraction $\sigma_0 = 0.5$ since the cost increased more quickly with σ_0. Finally, all of our online evaluations of the learned partial policies within UCT are averaged across 1000 games and 95% confidence intervals are plotted.

FT-QCM v OPI. Figures 2(a) and 2(d) compare the search performance of UCT given various amounts of time per move, when using the partial policies learned by FT-QCM to that learned by OPI. FT-QCM has clear wins across most of the anytime curve, with OPI achieving parity at sufficiently large budgets in both domains. In results not shown for space reasons, we repeated this experiment for a variety of sets of the pruning fraction σ_d. In all cases, OPI never outperforms FT-QCM for any setting of σ_d and at best, achieves parity. Based on this result the remainder of our experimental evaluation is based on FT-QCM.

FT-QCM v baselines. We now compare the performance of pruning with FT-QCM partial policies to other approaches for injecting control knowledge into UCT. Most other approaches require an action scoring function of some form, for which we use the ranking function f_0 learned for the root partial policy by FT-QCM. The first baseline is *vanilla* UCT without any knowledge injection. Second, we use (decaying) heuristic bias (*HB*) in UCT as discussed in the related work, using the best settings found by experimenting with different methods of setting the bias component. Our third baseline uses a softmax policy, parameterized by f_0, as the default policy (*DP*). During a rollout, an action is sampled with probability $\exp(f_0(s,a))/\sum_{a' \in A(s)} \exp(f_0(s,a'))$ instead of uniform random selection. Our fourth and last baseline, *greedy*, does not search and selects root actions greedily according to the learned root ranking function. Note that this final baseline can be viewed as an attempt to apply traditional imitation learning to our problem.

Figures 2(b) and 2(e) give the results for Galcon and Yahtzee. For Galcon, which is two-player, the curves correspond to each method playing against *vanilla* UCT using the same budget on wall-clock time. The heuristic bias (*HB*) technique is unable to show any improvement over *vanilla* when evaluated in terms of performance versus wall-clock time. This is likely due to f_0 being inaccurate and highly biased when used as an evaluation function. Furthermore, tuning the complex interaction of Q estimates, exploration bonus and noisy bias terms is challenging and time-consuming. Note that *HB* does not decrease search performance unlike the informed default policy (*DP*) baseline. Here, the cost of using the expensive softmax rollout policy dwarfs any benefit. *DP* only seems to "pay for itself" at the largest budgets where parity is

[2] When UCT generates tree nodes at depths greater than D, in these experiments we prune using ψ_D. However, we have confirmed experimentally that typically such nodes are not visited frequently enough for pruning to have a significant impact.



(a) FT-QCM v OPI (Galcon)

(b) FT-QCM v baselines (Galcon)

(c) # of simulations (Galcon)

(d) FT-QCM v OPI (Yahtzee)

(e) FT-QCM v baselines (Yahtzee)

(f) # of simulations (Yahtzee)

Figure 2: The first column shows a comparison between partial policies learned by FT-QCM and OPI when used inside UCT for different time bounds. In (a), the baseline player is OPI instead of *vanilla* (b,c). The second column shows the main experimental result where *FT-QCM* consistently outperforms baselines in both domains. *DP* and *HB* are surprisingly bad when knowledge costs are taken into account. The third column shows performance under zero knowledge costs by measuring the budget in simulation counts. Again, *FT-QCM* does well consistently but now *DP* and *HB* perform better.

achieved with the other baselines. The last baseline *greedy* demonstrates the limited strength of the reactive policy which is only competitive in Galcon when the search is very time-constrained. In Yahtzee, *greedy* is not competitive at any budget. We note that we have attempted to apply alternative imitation learning approaches to arrive at improved reactive policies, with little improvement. It appears that our feature representation is not powerful enough to accurately discriminate optimal actions. By relaxing our goal to learning a partial policy and integrating with search, we are able to benefit significantly from learning.

In both domains, FT-QCM outperforms every baseline at every budget considered. In Galcon, it has large wins on the left of the anytime curve and smaller wins on the right (not visible due to scale). In Yahtzee as well, FT-QCM achieves higher scores consistently. Furthermore, it continues to improve even at the largest budgets. In Yahtzee, increasing average score above 200 is extremely challenging and FT-QCM's performance improvement is significant.

The cost of knowledge. It is initially surprising that the heuristic bias and informed default policy methods do not improve over *vanilla* MCTS. It turns out these methods do well as long as the cost of using knowledge is set to zero. That is, budgets are specified in terms of the number of simulations conducted by UCT instead of wall-clock time. The difference is shown in Figures 2(c) and 2(f) where all the planners were re-run using simulations instead of time. Now *HB* outperforms *vanilla* in both domains while *DP* is competitive or wins by small amounts. However, our method FT-QCM is again better across most of the anytime curve in both domains. This result shows the importance of evaluating in terms of wall-clock time, which has not always been common practice in prior work.

7 SUMMARY

We have shown algorithms for offline learning of partial policies for reducing the action branching factor in time-bounded tree search. The algorithms leverage a reduction to i.i.d supervised learning and are shown to have bounds on the worst case regret. Experiments on two challenging domains show significantly improved anytime performance in Monte-Carlo Tree Search. One line of future work involves more sophisticated uses of partial policies during search (e.g., progressive widening, time-aware search control) for further improvements in anytime performance. We are also interested in iterating the learning process, where we use a learned ψ to guide deeper search in order to generate training data for learning an even better ψ.

Acknowledgements

This work was supported by NSF grant IIS-1219258.

References

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *MLJ*, 47(2-3):235–256, May 2002.

Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *AI*, 72(1-2):81–138, 1995.

Blai Bonet and Hector Geffner. Action selection for MDPs: Anytime AO* versus UCT. In *AAAI*, 2012.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *CIG*, 4(1):1–43, 2012.

Vadim Bulitko and Greg Lee. Learning in real-time search: A unifying framework. *JAIR*, 25:119–157, 2006.

Guillaume Chaslot, Mark Winands, Jaap H van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for Monte-Carlo tree search. pages 655–661, 2007.

Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *LION*, pages 433–445. Springer, 2011.

Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *ICML*, pages 273–280. ACM, 2007.

Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical report, INRIA, 2006.

Todd Hester and Peter Stone. Texplore: real-time sample-efficient reinforcement learning for robots. *MLJ*, 90(3): 385–429, 2013.

Michael Kearns, Yishay Mansour, and Andrew Y Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. *MLJ*, 49(2-3):193–208, 2002.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, pages 282–293. 2006.

John Langford. Vowpal Wabbit. *URL https://github.com/JohnLangford/vowpal_wabbit/wiki*, 2011.

Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *AISTATS*, pages 661–668, 2010.

Stéphane Ross, Geoffrey J Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, pages 627–635, 2011.

David Silver and Gerald Tesauro. Monte-Carlo simulation balancing. In *ICML*, pages 945–952, 2009.

Jonathan Sorg, Satinder P Singh, and Richard L Lewis. Optimal rewards versus leaf-evaluation heuristics in planning agents. In *AAAI*, 2011.

Umar Syed and Robert E Schapire. A reduction from apprenticeship learning to classification. In *NIPS*, pages 2253–2261, 2010.

Joel Veness, David Silver, Alan Blair, and William W Cohen. Bootstrapping from game tree search. In *NIPS*, pages 1937–1945, 2009.

Joel Veness, Kee Siong Ng, Marcus Hutter, William Uther, and David Silver. A Monte-Carlo AIXI approximation. *JAIR*, 40(1):95–142, 2011.

Thomas J. Walsh, Sergiu Goschin, and Michael L. Littman. Integrating sample-based planning and model-based reinforcement learning. In *AAAI*, 2010.

Yuehua Xu, Alan Fern, and Sungwook Yoon. Learning linear ranking functions for beam search with application to planning. *JMLR*, 10:1571–1610, December 2009.

Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *JMLR*, 9:683–718, 2008.