Journal of Machine Learning Research 18 (2017) 1-35 Submitted 5/15; Revised 10/16; Published 8/17

Learning Partial Policies to Speedup MDP Tree Search via Reduction to I.I.D. Learning

Jervis Pinto
JERVIS.PINTO@GMAIL.COM
School of EECS, Oregon State University, Corvallis OR 97331 USA

Alan Fern
AFERN@EECS.OREGONSTATE.EDU
School of EECS, Oregon State University, Corvallis OR 97331 USA

Editor: Michael Bowling

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 consists of reducing the learning problem to set learning. We give a reduction-style analysis of three such algorithms, each making different assumptions, which relates the set 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 on two challenging MDPs with large action branching factors. The results show that the learned partial policies can significantly improve the anytime performance of Monte-Carlo tree search.

Keywords: online sequential decision-making, partial policy, partial policy learning, imitation learning, Monte-Carlo tree search, reductions

1. Introduction

Online sequential decision-making involves selecting actions in order to optimize a long-term performance objective while meeting practical decision-time constraints. In many environments, including Chess, Go, computer networks, automobiles and aircraft, it is possible to obtain simulators or learn an accurate model of the environment, whereas in others, no such simulator or model is available. A common approach in both cases is to use learning techniques such as reinforcement learning (RL) (Sutton and Barto, 1998) or imitation learning (Pomerleau, 1989; Sammut et al., 1992) in order to learn reactive policies, which can be used to quickly map any state to an action. While reactive policies support fast decision making, for many complex problems, it can be difficult to represent and learn high-quality reactive policies. For example, in games such as Chess, Go, and other complex problems, the best learned reactive policies are significantly outperformed by the best approaches based on look-ahead search (Gelly and Silver, 2007; Veness et al., 2009; Silver et al., 2016). As another example, recent work has shown that effective reactive policies can be learned for playing a variety of Atari video games using both RL (Mnih et al., 2013) and imitation learning (Guo et al., 2014). However, the performance of these reactive policies are easily outperformed or equaled by a vanilla lookahead planning agent (Guo et al., 2014).

Faced with the choice of a planning agent versus a (model-free) learning agent, a practitioner must consider two key factors. The first is the non-trivial expense of obtaining a simulator or learning a model. The second is that the planning agent requires significantly more time per decision. That is, even when a simulator is available, the computational expense may not be practical for certain applications, including real-time Atari game play.

This paper attempts to make progress in the region between fast reactive decision making and slow lookahead search, when a simulator or model of the world is available. In particular, we study algorithms for learning to improve the anytime performance of lookahead search, allowing for more effective deliberative decisions to be made within practical time bounds. Lookahead tree search constructs a finite-horizon lookahead tree using a (possibly stochastic) model or simulator in order to estimate action values at the current state. 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* (Blai and Hector, 2012). However, under practical time constraints (e.g., one second per root decision), the performance of these approaches strongly depends on the action branching factor. In non-trivial MDPs, the number of possible actions at each state is often considerable, which 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 the offline learning of such prior knowledge in the form of partial policies [1].

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 the original search method that does not prune.

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 set learning problems, characterized by choices for (1) and (2). The set learning problems are further reduced to cost-sensitive binary classification. 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

1. A preliminary version of this work appeared in UAI 2014 (Pinto and Fern, 2014).

reduction-style results for learning (complete) policies via imitation (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) a real-time strategy game, Galcon and 2) a classic dice game, Yahtzee. 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). An MDP is a tuple (S, A, P, R), where S is a finite set of states, A is 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. In environments where the reward is non-deterministic (but still stationary), $R(s, a)$ is taken to be the expected value. The typical goal in 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. For example, two popular choices are maximizing the expected finite-horizon total reward or expected infinite-horizon discounted reward.

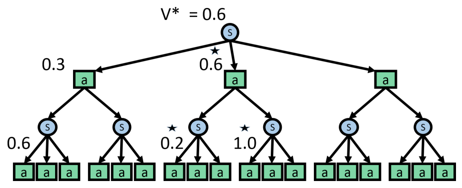
In practice, regardless of the long-term reward measure, for large MDPs, the offline computation of high-quality policies over all environment states is impractical. In such cases, a popular action-selection approach is online tree search, where at each encountered environment state, a time-bounded search is carried out in order to estimate action values. Note that this approach requires the availability of either an MDP model or a 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.

2.1 Online Tree Search

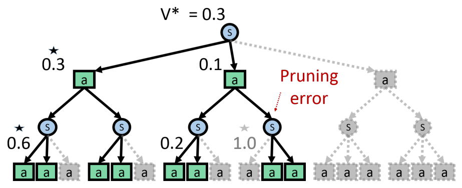
Throughout this paper, we will focus on search that has a depth bound D. Doing so bounds the length of future action sequences to be considered. Given a state s, we denote by $T(s)$ the depth D expectimax tree rooted at s. $T(s)$ alternates between layers of state nodes and action nodes[2], labeled by 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. Note 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. $V^*_d(s)$ and $Q^*_d(s, a)$ are formally defined next.

2. Also known as "chance" nodes in the context of expectimax trees.



(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. The distributions over the transitions to the next state are uniform. The numeric values are the optimal action values Q^*. Nodes for which no value is shown have $Q^*(s, a) = 0$. Nodes marked with an asterisk indicate that the node corresponds to the optimal action choice. In (b), pruning can change the value at the root if nodes on the optimal paths are incorrectly pruned.

$$\begin{aligned} V_d^*(s) &= \max_a Q_d^*(s, a) && \\ Q_d^*(s, a) &= R(s, a) && \text{if} \quad d = D - 1 \qquad (1) \\ &= R(s, a) + E_{s' \sim P(\cdot \mid s, a)} \left[V_{d+1}^*(s') \right] && \text{otherwise} \qquad (2) \end{aligned}$$

Equation 1 corresponds to leaf action nodes. In Equation 2, $s' \sim P(\cdot|s, a)$ ranges over the children of a. Given these value functions, the optimal action policy for state s at depth d is

$$\pi_d^*(s) = \arg\max_a Q_d^*(s, a)$$

Given an environment state s, online search algorithms such as UCT or RTDP attempt to completely or partially search the tree $T(s)$ in order to approximate the root action values $Q_0^*(s, a)$ well enough to identify the optimal action $\pi_0^*(s)$. 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 number of actions that will be taken in the actual environment (e.g., the length of a full game). 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.

2.2 Search with a Partial Policy

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. In this paper, we will utilize partial policies for pruning actions. 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. In this work, we focus exclusively on non-stationary partial policies, rather than also considering stationary partial policies where $\psi_i = \psi_j$ for all i and j. One reason for this is that optimal solutions to depth-bounded search trees are non-stationary in general. In addition, considering non-stationary policies allows for defining and analyzing relatively simple training algorithms as described in section 3. In some applications, however, where the planning depth is not known in advance, it may be desirable to learn stationary partial policies. Extending our work to the stationary scenario is a point of future work, which we expect can draw on ideas for learning stationary policies in imitation learning (Ross and Bagnell, 2010; Ross et al., 2011).

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 $a \notin \psi_d(s)$. The phrase "root state" is not to be confused with the starting state of a sequential problem (e.g., the initial board in Chess). Rather, "root state" refers to *any* state in which the search agent is required to act and must therefore place the state at the root of the search tree. 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 before and let $V_d^\psi(s)$ and $Q_d^\psi(s,a)$ denote the depth d state and action value functions, as follows.

$$\begin{aligned} V_d^\psi(s) &= \max_{a \in \psi_d(s)} Q_d^\psi(s,a) \\ Q_d^\psi(s,a) &= R(s,a) & \text{if} \quad d = D-1 \\ &= R(s,a) + E_{s' \sim P(\cdot \mid s,a)}\left[V_{d+1}^\psi(s')\right] & \text{otherwise} \end{aligned}$$

We will denote the highest-valued, or greedy, root action of $T_\psi(s)$ as

$$\pi^\psi(s) = \arg\max_{a \in \psi_0(s)} Q_0^\psi(s,a)$$

This is the root action that a depth D search procedure would attempt to return in the context of ψ. We say that a partial policy ψ subsumes a partial policy ψ' if for each s and d, $\psi'_d(s) \subseteq \psi_d(s)$. In this case, it is straightforward to show that for any s and d, $V_d^{\psi'}(s) \leq V_d^\psi(s)$. This is simply because planning under ψ allows for strictly more action choices than planning under 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. The second special case occurs at the other extreme where $\psi_d = A$ for all d. This corresponds to full-width search and we have $V^* = V^\psi$ and $Q^* = Q^\psi$.

Clearly, a partial policy can reduce the complexity of search by eliminating some actions. 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 $V_0^*(s) - Q_0^*(s,a)$. Note 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

$$\text{REG}(\mu_0, \psi) = E\left[V_0^*(s_0) - Q_0^*(s_0, \pi^\psi(s_0))\right] \quad \text{where } s_0 \sim \mu_0 \tag{3}$$

2.3 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. As an illustrative example, consider the domain of Chess, where we wish to learn a partial policy for pruning the search tree. Our approach is to use expensive search offline to play many full games of Chess and to record the sequence of states encountered during those games. Each state encountered during actual play is called a *root state* of a search tree. The collected root states and associated trees are then used to learn a partial policy that can be used to speedup future online search.

More formally, 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. In our Chess example, μ_0 would be a distribution over the states encountered along games played using the expensive search. 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. A typical choice of μ_0 is the distribution of states encountered along trajectories of a receding-horizon controller that makes decisions based on unpruned depth D search. We use this definition of μ_0 in our experiments. This setup is closely related to imitation learning (Syed and Schapire, 2010; Ross and Bagnell, 2010; Ross et al., 2011) since we are essentially treating unpruned, expensive search as the expert to be imitated by a pruned search. Other choices of the initial state distribution μ_0 are the state distributions that arise from more complex imitation learning algorithms such as DAGGER (Ross et al., 2011).

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 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 not prune. At the other extreme, we can remove the need for any search by learning a partial policy that always returns a single action, which corresponds to 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 may be much less.

In practice, we seek a good tradeoff between the two extremes. The tradeoff between a reactive policy versus a full depth D search-based policy is application-specific. 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.

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 *set learning* problems, each producing one partial policy component ψ_d. Informally, a supervised set learning problem is similar to a traditional classification problem, except that the learned functions (partial policies, in this case) return sets of labels rather than a single label. In our work, labels will correspond to actions and the designer will specify the size of the label/action sets that the learned function should return. The quality of a learned set function will be evaluated in terms of a cost function that provides a measure of whether a set returned by the function contains "good" labels/actions.

More formally, the set learning problem for partial policy component 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 will be a monotone decreasing set function of A' with zero pruning having zero cost, $C_d(s, A) = 0$. Note that a trivial solution to the learning problem, which achieves zero cost, would be to learn a function that always returns the complete action set A. However, such solutions will be avoided by having the designer specify the amount of pruning that they desire, or equivalently the size of the action sets returned. Note that constraining the set to contain exactly one action would correspond to learning complete policies.

We begin by assuming the availability of a set learning algorithm called SETLEARN that takes three inputs, namely, a set of states drawn from μ_d, cost function C_d, and a pruning percentage $0 < \sigma_d < 1$. The output of SETLEARN is a partial policy component ψ_d that returns action sets that contain at most a fraction $1 - \sigma_d$ of the available actions while attempting to minimize the expected cost of ψ_d on μ_d. That is, the goal is to return a ψ_d that minimizes $E\left[C_d(s, \psi_d(s))\right]$ for $s \sim \mu_d$. The designer's choice of pruning percentage σ_d (e.g., prune 75% of the actions in each state) is related to the time constraints of the decision problem, with smaller time constraints requiring more pruning.

Given access to SETLEARN we first present our generic reduction algorithm for learning partial policies in Algorithm 1. This algorithm template simply calls SETLEARN on a sequence of

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 $d \in \{0, \ldots, D-1\}$. SETLEARN is a set learning algorithm that aims to minimize the expected cost of each ψ_d relative to C_d and μ_d. σ_d is a pruning fraction such that SETLEARN returns at most $1 - \sigma_d$ actions. Each partial policy ψ_d is a set function that maps states to action sets of size $(1 - \sigma_d)|A|$.

```
1: procedure PARTIALPOLICYLEARNER($\{(\mu_d, C_d)\}, \sigma_d$)
2:     for $d = 0, 1, \ldots, D-1$ do
3:         Sample a training set of states $S_d$ from $\mu_d$
4:         $\psi_d \leftarrow$ SETLEARN($S_d, C_d, \sigma_d$)
5:     end for
6:     return $\psi = (\psi_0, \psi_1, \ldots, \psi_{D-1})$
7: end procedure
```

set learning problems and returns a list of the learned partial policies. In order to instantiate this template, it is necessary to specify SETLEARN and the individual set learning problems (μ_d, C_d). We defer details of our implementation of SETLEARN until section 4. Rather, we proceed with the definition of our learning reductions. Each reduction is specified by a particular choice of (μ_d, C_d), such that we can bound the expected regret of ψ (when used for search) by the expected (i.i.d.) costs of the ψ_d returned by SETLEARN.

We begin with the state distributions μ_d. These are specified in terms of distributions induced by (complete) policies. In particular, given a policy π, we let $\mu_d(\pi)$ denote the state distribution produced by the following procedure: Draw an initial state from μ_0, execute π for d steps and return the final state. 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 two simple lemmas that will be used to prove our regret bounds.

Lemma 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 $s_0 \sim \mu_0$.*

Proof Since π is subsumed by ψ, we know that $Q_0^\psi(s, \pi^\psi(s)) = V_0^\psi(s) \geq V_0^\pi(s)$. Since for any a, $Q_0^*(s,a) \geq Q_0^\psi(s,a)$, we have for any state s, $Q_0^*(s, \pi^\psi(s)) \geq V_0^\pi(s)$. 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. Our three reduction algorithms, presented below, provide different approaches for making such a guarantee.

The second lemma will be used to bound the sub-optimality of complete polices that are subsumed by our learned partial policies, which then allows the application of Lemma 1. Versions of this "performance difference lemma" have been used in prior work (Kakade and Langford, 2002; Bagnell et al., 2003; Ross and Bagnell, 2014).

Lemma 2 *For any two complete policies π and π' and any initial state distribution μ_0,*

$$E\left[V_0^{\pi'}(s_0) - V_0^\pi(s_0)\right] = \sum_{d=0}^{D-1} E\left[V_d^{\pi'}(s_d) - Q_d^{\pi'}(s_d, \pi(s_d))\right], \textit{where } s_d \sim \mu_d(\pi_{0:d-1})$$

.

Proof Following a prior proof (Ross and Bagnell, 2014), define π_d to be a policy that follows π for d steps and then at step $d+1$ switches to following π' until the maximum depth D. Using the observations that $\pi_0 = \pi'$ and $\pi_D = \pi$, we can derive the following.

$$\begin{aligned} E\left[V_0^{\pi'}(s_0) - V_0^\pi(s_0)\right] &= E\left[\sum_{d=0}^{D-1} V_0^{\pi_d}(s_0) - V_0^{\pi_{d+1}}(s_0)\right] \\ &= \sum_{d=0}^{D-1} E\left[V_0^{\pi_d}(s_0) - V_0^{\pi_{d+1}}(s_0)\right] \\ &= \sum_{d=0}^{D-1} E\left[V_d^{\pi'}(s_d) - Q_d^{\pi'}(s_d, \pi(s_d))\right], \text{where } s_d \sim \mu_d(\pi_{0:d-1}) \end{aligned}$$

The first equality is simply a telescoping sum. The third equality follows due the fact that π_d and π_{d+1} both follow π for the first d steps, which yields the same distribution over s_d. ■

We will use this lemma when the reference policy π' is optimal, that is, $\pi' = \pi^*$. Thus, the sub-optimality of π can be bounded by accumulating the expected sub-optimality of its action choices along trajectories generated by π. Next, we present our three reduction algorithms, which are summarized in Table 1.

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

OPI	
	$\mu_d = \mu_d(\pi^*)$
	$C_d(s, \psi_d(s)) = 0$ if $\pi^*_d(s) \in \psi_d(s)$, otherwise 1
FT-OPI	
	$\mu_d = \mu_d(\psi^+_{0:d-1})$
	$C_d(s, \psi_d(s)) = 0$ if $\pi^*_d(s) \in \psi_d(s)$, otherwise 1
FT-QCM	
	$\mu_d = \mu_d(\psi^*)$
	$C_d(s, \psi_d(s)) = Q(s, \pi^*_d(s)) - Q(s, \psi^*_d(s))$

Table 1: Instantiations for OPI, FT-OPI and FT-QCM in terms of the template in Algorithm 1. Note that ψ_d is a partial policy while π^*_d, ψ^+_d and ψ^*_d are complete policies.

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. We refer to this as the pruning error and denote it by

$$e^*_d(\psi) = \Pr_{s \sim \mu_d(\pi^*)} (\pi^*_d(s) \notin \psi_d(s)) \tag{4}$$

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. This is an important point since we will approximate π^* using long, computationally expensive runs of UCT, described in section 4.

Intuitively, 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 proof 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 *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 $d \in \{0, \ldots, D-1\}$. Given this constraint on ψ^+, we can apply Lemma 3 from (Syed and Schapire, 2010)[3], which implies $E[V_0^{\psi^+}(s_0)] \geq E[V_0^*(s_0)] - \epsilon D^2$, where $s_0 \sim \mu_0$. The result follows by combining this with Lemma 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. Borrowing again from imitation learning, 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 each ψ_d returned by SETLEARN attempts to maximize 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 $T(s_0)$. Now suppose 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 node n.

In imitation learning, this type of problem has been dealt with via "forward training" of non-stationary policies (Ross et al., 2011). We employ a similar idea in the Forward Training OPI (FT-OPI) algorithm. FT-OPI 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 $d = D - 1$. Each ψ_d is trained 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 in Algorithm 1 are $\mu_d = \mu_d(\psi^+_{0:d-1})$ and the cost function remains zero-one cost as for OPI. Thus, the expected cost of ψ_d is $e^+_d(\psi) = \Pr_{s \sim \mu_d(\psi^+_{0:d-1})}(\pi^*_d(s) \notin \psi_d(s))$, 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 from 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

3. 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.

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 4 *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 We first bound the sub-optimality of ψ^+. By applying Lemma 2 with $\pi' = \pi^*$ and $\pi = \psi'$ we can infer

$$E\left[V_0^*(s_0) - V_0^{\psi^+}(s_0)\right] = \sum_{d=0}^{D-1} E\left[V_d^*(s_d) - Q_d^*(s_d, \psi^+(s_d))\right], \text{where } s_d \sim \mu_d(\psi^+_{0:d-1}). \quad (5)$$

Since we have assumed that $e_d^+(\psi) \leq \epsilon$, we know that for each d the probability that ψ^+ does not select an optimal action for states $s_d \sim \mu_d(\psi^+_{0:d-1})$ is no more than ϵ. In addition, by assumption, the worst case regret of such erroneous action choices is bounded by Δ. Thus, each expectation term of the right-hand-side of Equation 5 can be bounded by $\epsilon\Delta$. Summing these D terms then implies that

$$E\left[V_0^*(s_0) - V_0^{\psi^+}(s_0)\right] \leq \epsilon \Delta D.$$

Since ψ^+ is subsumed by ψ, we can apply Lemma 1 to yield the result. ■

This result implies that if Δ is significantly smaller than D, then 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 zero-one cost is its inability to distinguish between large pruning errors and small pruning errors. 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. For example, in Chess, some actions might lead to unavoidable defeat. 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 has small zero-one cost while also providing 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. We use long runs of UCT to approximate ψ^*_d and use UCT's Q estimates in the cost function. The full implementation is described in section 4.

There is a significant theoretical benefit to using expected Q-cost for learning compared to zero-one cost. The following bound, which motivates the FT-QCM algorithm, shows that the dependence on D decreases from worst-case quadratic (for OPI and FT-OPI) to linear.

Theorem 5 *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$(\mu_0, \psi) \leq \Delta D$.*

Proof Applying Lemma 2 with $\pi' = \pi^*$ and $\pi = \psi^*$ we get that

$$E\left[V^*_0(s_0) - V^{\psi^*}_0(s_0)\right] = \sum_{d=0}^{D-1} E\left[V^*_d(s_d) - Q^*_d(s_d, \psi^*(s_d))\right], \text{where } s_d \sim \mu_d(\psi^*_{0:d-1}).$$

By our assumption that $\Delta_d(\psi) \leq \Delta$ for all d, we can combine this with the above to obtain

$$E\left[V^*_0(s_0) - V^{\psi^*}_0(s_0)\right] \leq \Delta D.$$

Since ψ^* is subsumed by ψ we can apply Lemma 1 to get the result. ■

FT-QCM tries to minimize this regret bound by minimizing $\Delta_d(\psi)$ via supervised learning at each depth. As we will show in our experiments, it is 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. When this is true, the benefits of FT-QCM over OPI and FT-OPI can be significant, which is shown in our experiments.

4. Implementation Details

This section first describes the UCT algorithm, which is the base planner used in all the experiments. We then describe the partial policy representation and the learning algorithm. Finally, we specify how we generate the training data.

4.1 UCT

UCT (Kocsis and Szepesvári, 2006) is an online planning algorithm. Given the current state s, UCT selects an action by building a sparse lookahead tree over the state space reachable from s. Thus, s is at the root of the tree. Edges from s correspond to actions and their outcomes so that the tree consists of alternating layers of state and action nodes. Finally, leaf nodes correspond to terminal states. Each state node in the tree stores Q estimates for each of the available actions. During search, the Q values are used to select the next action to be executed. The algorithm is anytime, which means that a decision can be obtained by querying the root at any time. A common practice is to run the algorithm for a fixed number of trajectories and then select the root action which has the largest Q value.

UCT became famous for advancing the state-of-the-art in Computer Go (Gelly et al., 2006; Gelly and Silver, 2007). Since then, however, many additional successful applications have been reported, including but not limited to the IPCC planning competitions (Keller and Helmert, 2013), general game playing (Méhat and Cazenave, 2010; Finnsson, 2012), Klondike Solitaire (Bjarnason et al., 2009), tactical battles in real-time strategy games (Balla and Fern, 2009) and feature selection (Gaudel and Sebag, 2010). See Browne et al. (2012) for a comprehensive survey.

The algorithm is unique in the way that it constructs the tree and estimates action values. Unlike standard minimax search or sparse sampling (Kearns et al., 2002), which build depth-bounded trees and apply evaluation functions at the leaves, UCT neither imposes a depth bound nor does it require an evaluation function. Rather, UCT incrementally constructs a tree and updates action values by carrying out a sequence of Monte-Carlo rollouts of entire trajectories from the root to the terminal state. The key idea in UCT is to bias the rollout trajectories towards promising ones, indicated by prior trajectories, while continuing to explore. The outcome is that the most promising parts of the tree are grown first while still guaranteeing that an optimal decision will be made given sufficient time and memory. Tree construction is incremental since each rollout introduces only a small number of additional nodes to the tree, typically one state node and its immediate children (action nodes).

There are two key algorithmic choices in UCT. The first is the policy used to conduct each rollout. We employ the popular method of random action selection which is simple, fast, and requires no domain knowledge. The second choice is the method for updating the value estimates in the tree in response to the outcome produced by the rollout. A commonly used technique requires that each node store: a) the number of times the state (or action) node has been visited in previous rollouts $n(s)$ (or $n(s,a)$), b) the current estimate of each action value $Q(s,a)$. Given this information at each state node, UCT performs a rollout starting at the root state of the tree. If there exist actions that have yet to be tried, then a random choice is made over these untried actions. Otherwise, when all actions have been tried at least once, the algorithm selects the action that maximizes an upper confidence bound given by

$$Q_{UCB}(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}} \tag{6}$$

where c is a positive constant that controls the exploration bonus. In practice, c is often tuned separately on a per-domain basis. The selected action is then executed and the resulting state is added to the tree if it is not already present. This action selection rule is based on the UCB rule (Auer et al., 2002) for multi-armed bandits, which seeks to balance exploration and exploitation. The first term $Q(s,a)$ is large for actions with high Q value estimates (exploitation), while the second term is large for actions that have been visited less often (exploration), relative to the number of visits to the parent state. The exploration bonus decreases as the action is visited more often.

Having described a procedure to generate trajectories, we now describe the method used to update the tree. After the trajectory reaches a terminal state and receives a reward R, the action values and counts of each state along the trajectory are updated. In particular, for any state action pair (s, a), the update rules are,

$$n(s) \leftarrow n(s) + 1 \tag{7}$$

$$n(s, a) \leftarrow n(s, a) + 1 \tag{8}$$

$$Q(s, a) \leftarrow Q(s, a) + \frac{1}{n(s, a)}(R - Q(s, a)) \tag{9}$$

The equation computes the average reward of rollout trajectories that pass through the node (s, a). As previously mentioned, the algorithm is run for some fixed number of simulations before selecting the best action at the root, typically, the action with the largest Q value, although selecting the action with the most visits is also popular. This particular algorithm has two parameters, the exploration constant c, and the number of rollout trajectories.

The above version is the most basic form of UCT, which has received a very large amount of attention in the game-playing and planning community. See Browne et al. (2012) for a comprehensive survey of the MCTS family of algorithms. We defer the discussion of the MCTS family and relevant UCT variants to section 5. Next, we describe the partial policy representation and learning framework used in our experiments.

4.2 Partial Policy Representation and Learning

We begin by reducing partial policy learning to set learning. As mentioned before, the designer chooses a fixed fraction of actions to prune in each state. The pruning fraction σ_d can vary from 0 to 1, where $\sigma_d = 1$ indicates maximum pruning. The choice of σ_d allows the designer to tradeoff the time constraints of the decision problem with the expected cost achievable by the SETLEARN algorithm. In this work, we perform a post-hoc analysis of the expected cost of ψ_d for a range of pruning values and select values of σ_d that yield reasonably small costs. In section 6, we give details of the selections used in our experiments.

Once σ_d is fixed, we have induced a set learning problem. Next, we convert the set learning problem to a ranking problem. That is, for states at depth d, we seek to learn a scoring function over actions that will allow us to prune away the bottom σ_d fraction of actions. In this work, we use linear ranking functions for their computational efficiency and simplicity, although non-linear approaches (e.g., deep neural networks) may also be used. Formally, we have a n-dimensional weight vector w_d and a user-provided feature vector $\phi(s, a)$ over state-action pairs. The linear ranking function $f_d(s, a) = w_d^T \phi(s, a)$ allows us to define a total order over actions, breaking ties lexicographically. Thus, w_d along with the pruning fraction σ_d fully parameterizes the partial policy component ψ_d, which is defined as the set of $\lceil (1 - \sigma_d) \times |A| \rceil$ highest ranked actions.

Next, we specify how we implement the SETLEARN 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. We first describe the reduction to cost-sensitive binary classification and then specify the method by which the training data is generated from long runs of UCT. 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, obtained from the Q estimates during offline UCT. We learn the partial policy by first learning the ranking function in a way that attempts to rank the optimal action as highly as possible and then select an appropriate pruning fraction 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) with optimal action a^*, we create a cost-sensitive classification example for each action $a_j \neq a^*$ of the form,

$$(s, c) \quad \longrightarrow \quad \{(\phi(s, a^*) - \phi(s, a_j)\ ,\ c(a^*) - c(a_j)) \quad | \quad a_j \neq a^*\}$$

Learning a linear classifier for such an example will attempt to rank a^* above a_j according to the cost difference. This produces $m - 1$ training examples for a state with m actions. We apply an existing cost-sensitive learner (VW, (Langford, 2011)) to learn a weight vector based on the pairwise data. The learning algorithm is standard binary classification with squared loss and L_2 regularization, set to 0.01. We also randomly flip the ranking constraints with uniform probability in order to balance the binary class labels. In VW format, each training example consists of a cost, binary label, and feature difference vector. Finally, we experimented with other methods of generating ranking constraints for the reduction from set learning to ranking (e.g., all pairs, unpruned vs pruned). The simple technique described above performed best in this experimental setup. Note that this implies that the training of f_d is independent of σ_d.

4.3 Generating Training States

Each of our algorithms requires sampling training states from trajectories of particular policies. OPI and FT-OPI require approximately computing π_d^*. FT-QCM has the additional requirement of approximating the action values for sub-optimal actions. 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. We then sample trajectories from those trees.

More specifically, our learning algorithm is provided with a set of root states by using expensive runs of UCT to select actions along complete trajectories, starting from a state drawn from the initial state distribution of a domain. We let S_0 denote the set of states encountered along these trajectories, noting that we can view S_0 as being drawn from the target distribution μ_0. For each $s_0 \in S_0$, we store the tree produced by UCT, noting that the resulting trees will typically have a large number of nodes on the tree fringe that have been infrequently visited. Since such states are not useful for learning, we select a depth bound D such that nodes at depth $d < D$ have been sufficiently explored and have meaningful action values. The trees are then pruned to depth D.

Given this set of depth D trees, we can now generate execution trajectories using the action selection rule for each algorithm (Table 1) and the MDP simulator to sample subsequent states. For example, OPI simply requires running trajectories through the trees based on selecting actions according to the optimal action estimates at each tree node. 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 highest-valued action 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 assumes i.i.d. training sets for each ψ_d, we only sample a single trajectory from each tree. However, our experiments indicate that multiple trajectories can be sampled from the tree without significant performance loss (although the guarantee is lost). Doing so allows many additional examples to be generated per tree, avoiding the need for additional expensive tree construction.

Algorithm	Action cost vector	Sample trajectory
OPI	$[..., I_{a^*=a_i}, ...]$	$s_0 \xrightarrow{\pi_0^*(s_0)} s_1 \xrightarrow{\pi_1^*(s_1)} s_2$ ...
FT-OPI	$[..., I_{a^*=a_i}, ...]$	$s_0 \xrightarrow{\psi_0^+(s_0)} s_1 \xrightarrow{\psi_1^+(s_1)} s_2$ ...
FT-QCM	$[... , Q_d^*(s, a^*) - Q_d^*(s, a_i), ...]$	$s_0 \xrightarrow{\psi_0^*(s_0)} s_1 \xrightarrow{\psi_1^*(s_1)} s_2$ ...

Figure 2: Training data generation and ranking reduction for each algorithm. Start by sampling s_0 from the same initial state distribution μ_0. OPI uses the optimal policy at each depth whereas FT-OPI and FT-QCM use the most recently learned complete policy (at depth $d-1$). OPI and FT-OPI use 0-1 costs. Only FT-QCM uses the Q cost function, which turns out to be critical for good performance.

5. Related Work

While there is a large body of work on integrating learning and planning, 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. Examples include 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, which is the focus of our work.

Work on learning search control knowledge in deterministic planning and games is more related. One research direction has been on learning knowledge for STRIPS-style deterministic planners. Examples of this approach are 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 leaf evaluation heuristics has also been studied in the context of deterministic real-time heuristic search (Bulitko and Lee, 2006). As another example, evaluation functions for game tree search have been learned from the “principle variations” of deep searches (Veness et al., 2009). A related body of work involves iterative deepening heuristic search and its variants (Korf, 1985; Reinefeld and Marsland, 1994). Control knowledge is typically leveraged using heuristic scoring functions, transposition tables and principal variations. These methods often improve search by using memory-intensive control knowledge (e.g., hash tables) and incrementally expanding the search, but do not provide theoretical guarantees. Nevertheless, the idea of progressive widening and iterative deepening is extremely appealing and one that we intend to explore more in future work.

The idea of using deep UCT search to generate high-quality state-action training pairs has been proposed recently (Guo et al., 2014). This work is intuitively similar to the work in this paper. The high-quality training data is used to train a stationary reactive policy using a deep neural network (DNN) policy. This is standard imitation learning, where the goal is to learn a reactive policy. The resulting DNN policy performs very well on a number of ATARI games. However, even this sophisticated reactive policy cannot outperform deep (but slower) UCT search. This work provides examples of domains in which an advanced reactive controller is outperformed by sufficiently deep lookahead tree search. Our work attempts to provide a rigorous middle ground between reactive decision-making and expensive unpruned search.

There have been a number of efforts for utilizing domain-specific knowledge in order to improve/speedup MCTS, many of which are covered in recent surveys (Browne et al., 2012; Gelly et al., 2012). The two main methods involve progressive bias and progressive widening. The core idea of progressive bias involves adding a quantity $f(s, a)$ for guiding action selection during search. $f(s, a)$ may be hand-provided (Chaslot et al., 2007), or learned (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 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. The second method, progressive widening, is more popular in problems with large or continuous action spaces. The main idea here is to start the search with small action sets and introduce new actions as search progresses (Couëtoux et al., 2011a,b). For example, one may add a new action (sometimes called "unpruning") when the number of visits to a state exceeds a heuristically chosen threshold. Although progressive widening is intuitively appealing and clearly required for anytime search in large action spaces, we were unable to find any existing method that leverages control knowledge or works in a principled manner.

Finally, MCTS methods often utilize hand-coded or learned rollout policies (sometimes called "default policies") to improve anytime performance. A very well-known example is MoGo (Gelly et al., 2006). While this approach has shown promise in specific domains such as Go, where the policies can be highly engineered for efficiency, we have found that the large computational overhead of a learned rollout policy makes its usage hard to justify. The key issue is that the use of any learned rollout policy requires the evaluation of a feature vector at each state (or state-action pair) encountered during the rollout. Rollouts are typically far longer than the depth of the search tree, especially during the initial part of the sequential problem. This results in a massive number of additional feature evaluations, far greater than the number of evaluations performed by the heuristic bias methods discussed above which only require a feature evaluation once for every new action node added to the tree. Thus, the use of learned rollout policies 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 perform a large-scale empirical evaluation of the partial policy learning algorithms on two challenging MDP problems. The first, Galcon, is a variant of a popular real-time strategy game, while the second is Yahtzee, a classic dice game. Both problems pose a serious challenge to MCTS algorithms, for different reasons. However, in both domains, UCT, enhanced with learned partial policies, is able to significantly improve real-time search performance over every baseline. The key experimental findings are listed next.

1. The search performance of UCT using a partial policy learned by the FT-QCM algorithm, denoted as UCT(FT-QCM), is significantly better than that of FT-OPI and OPI in both domains.

2. UCT(FT-QCM) performs significantly better than vanilla UCT and other "informed" UCT variants when the search budget is small. Given a larger search budget, UCT(FT-QCM) wins in Galcon and achieves parity in Yahtzee.

3. The average regret of the pruning policies on a supervised dataset of states is strongly correlated with the search performance, which is in line with our formal results. FT-QCM has significantly lower regret than FT-OPI and OPI.

4. The quality of the training data deteriorates quickly as depth d increases. Regret increases as d increases.

5. It is critical to incorporate the computational cost of using control knowledge into the search budget during evaluation. Ignoring the computational cost of knowledge can significantly change the perceived relative performance of different algorithms.

6.1 Experimental Domains

Galcon: The first domain is a variant of a popular two-player real-time strategy game, illustrated in Figure 3. The agent seeks to maximize its population by launching variable-sized population fleets from one of the planets currently under its control. The intention may be to either reinforce planets already controlled, attack an enemy planet, or capture an unclaimed planet. This produces a large action space ($\mathrm{O}\left(|\text{planets}|^2\right)$), where an action is defined as the triple (source planet, destination planet, fleet size). In our experiments, we use maps with 20 planets and three discrete levels of fleet size (small, medium, large). This often leads to game states where the action space contains hundreds of actions. The exact set of actions varies during the game as control over planets changes. State transitions are typically deterministic, unless a battle occurs. During each battle, a unit is randomly chosen from the two armies for termination. The battle ends when only one army remains. The maximum length of a game is restricted to 90 moves.

The search complexity caused by the large action branching factor is exacerbated by the relatively slow simulator and the length of the game. In this setting, UCT can only perform a small number of rollouts even when the search budget is as large as eight seconds per move. In fact, even at the largest search budgets, UCT only sparsely expands the tree beyond depths two or three. Note, however, that the rollout component of UCT is run to the end of the game and hence provides information well beyond these shallow depths. Finally, the second player or adversary is fixed to be a UCT agent with a budget of one second per move. All evaluations are performed against this fixed adversary. Note that at one second per decision, the fixed adversary effectively acts randomly, especially during the initial part of the game. However, further increase in the adversary's search budget is impractical because of the significant computational cost of evaluating search agents in this domain.

The agent's performance can be measured in terms of the final population difference. This value is zero when the game is tied, a large positive value when the agent wins convincingly and a negative value when the agent is much weaker than the adversary. The rollout policy used is random unless mentioned otherwise, while the exploration bias c is set to 0.75 in Galcon, after a brief grid search showed that search performance was relatively invariant to the exploration bonus as long as the parameter was set sufficiently high. The exploration bias was not optimized in favor of any of the three algorithms presented in this paper and is kept constant.

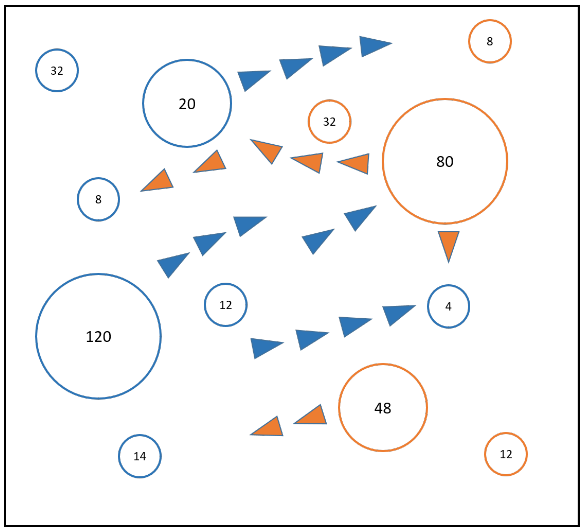
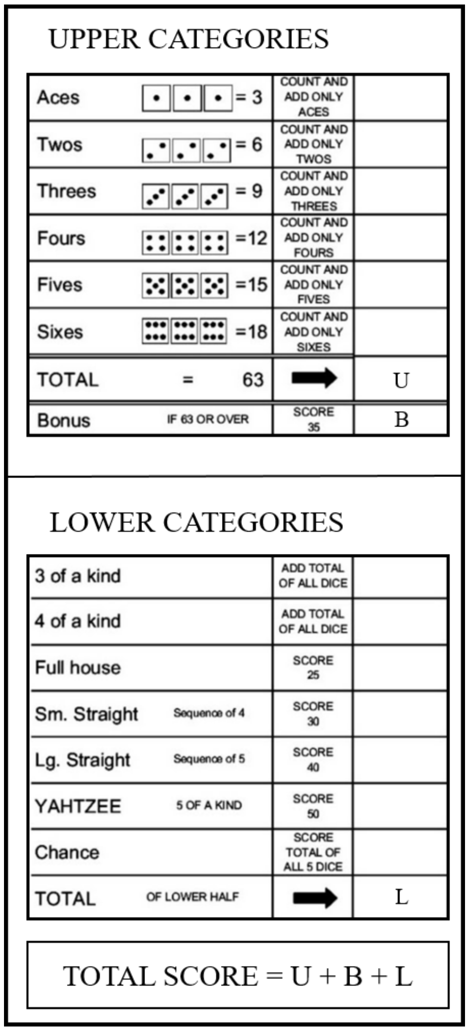


Figure 3: A visualization of a Galcon game in progress. The agent must direct variable-sized fleets to either reinforce planets under its control, take over enemy planets, or unoccupied ones. The decision space is large, often involving hundreds of actions.

Yahtzee: The second domain is a classic dice game which consists of thirteen stages. In each stage, the agent may roll any subset of the five dice at most twice, after which the player must select one of the empty categories. A category, once selected, may not be reselected in later stages. This produces a total of 39 decisions that must be made per game, where the last action trivially assigns the only remaining category. The categories are divided into upper and lower categories with each one scored differently. The lower categories roughly correspond to poker-style conditions (“full house”, “three of a kind”, “straight”, etc.), while the upper categories seek to maximize the individual dice faces (ones, twos, ..., sixes). A bonus of 35 points is awarded when the sum of the upper category scores exceeds 63. This ensures that the player must very carefully assign categories since the loss of the bonus has a large impact on the final score. The maximum score at the end of any game is 375, which is very hard to achieve. The above implementation is the simplest version of the game, restricted to a single “Yahtzee” (all five dice faces are identical). Our implementation for the simulator will be made available upon request[4]. Figure 4 shows a summary of the rules of the game. The UCT algorithm, including all parameter values, is identical to the one used in Galcon.

The trees generated by UCT differ significantly in the two domains. Compared to Galcon, Yahtzee has a relatively small action branching factor, smaller time horizon and a fast simulator. Furthermore, the available actions and the branching factor in Yahtzee decrease as categories are filled, whereas in Galcon, the agent occupying more planets has a larger action set. Thus, we can play many more games of Yahtzee per unit time compared to Galcon. However, Yahtzee poses a difficult challenge for MCTS algorithms due to the dice rolls which increase the state branching factor and add variance to the Q value estimates. Consequently, even deep search with the UCT algorithm is error-prone and struggles to improve performance with larger search budgets. As in Galcon, the search quality of UCT degrades significantly at depths beyond two or three. Note, however, that

4. We are also in the process of open-sourcing the entire code base.



UPPER CATEGORIES

Aces	⚀⚀⚀ = 3	COUNT AND ADD ONLY ACES	
Twos	⚁⚁⚁ = 6	COUNT AND ADD ONLY TWOS	
Threes	⚂⚂⚂ = 9	COUNT AND ADD ONLY THREES	
Fours	⚃⚃⚃ =12	COUNT AND ADD ONLY FOURS	
Fives	⚄⚄⚄ =15	COUNT AND ADD ONLY FIVES	
Sixes	⚅⚅⚅ =18	COUNT AND ADD ONLY SIXES	
TOTAL	= 63	➡	U
Bonus	IF 63 OR OVER	SCORE 35	B

LOWER CATEGORIES

3 of a kind		ADD TOTAL OF ALL DICE	
4 of a kind		ADD TOTAL OF ALL DICE	
Full house		SCORE 25	
Sm. Straight	Sequence of 4	SCORE 30	
Lg. Straight	Sequence of 5	SCORE 40	
YAHTZEE	5 OF A KIND	SCORE 50	
Chance		SCORE TOTAL OF ALL 5 DICE	
TOTAL	OF LOWER HALF	➡	L

TOTAL SCORE = U + B + L

Figure 4: A typical scoring sheet for Yahtzee, consisting of five dice and 13 categories. In each game, all 13 categories must be scored, one per stage. Each stage consists of at most three dice rolls (original roll + two more controlled by the player). The objective is to maximize the total score. The dice rolls increase the tree size and adds variance during rollouts. Our simulator implements the variant where at most one "Yahtzee" (all five dice faces are identical) can be recorded.

the performance degradation in Yahtzee is primarily caused by the transition stochasticity from the dice rolls, although the action branching factor remains significant. As in Galcon, UCT rolls out trajectories to the end of the game. Doing so provides information from states much deeper than the depth of the tree.

Normalizing rewards

The final score is always non-negative for Yahtzee, but may be negative for Galcon if the agent loses the game. In order to achieve a consistent comparison between the supervised metrics and search performance, we perform a linear rescaling of the score so that the performance measure now lies in the interval $[0, 1]$. In our experiments, we provide this measure as reward to the agent at the end of each trajectory. That is, rewards are zero everywhere except in terminal state-action pairs. Note that a reward above 0.5 indicates a win in Galcon. In Yahtzee, a reward value of 1.0 is the theoretical upper limit achievable in the best possible outcome, which rarely occurs in practice. Furthermore, even a small increase in the reward (e.g., 0.1) is a significant improvement in Yahtzee.

6.2 Partial Policy Learning Setup

We now describe the procedure used to generate training data for the three learning algorithms, OPI, FT-OPI, and FT-QCM. Each algorithm is provided with root states generated by playing approximately 200 full games. In Galcon, UCT is allowed to search for 600 seconds per move, whereas in Yahtzee, UCT is given 120 seconds per move. Each game results in a trajectory of states visited during the game. All of those states across all games constitute the set of root states used for training as described in section 4. Despite the enormous search budget, depths beyond two or three are sparsely expanded, due to the large state and action branching factors. We therefore set the search depth bound (D) to 3 for learning, since the value estimates produced by UCT for deeper nodes are often inaccurate. Thus, the learned partial policy has the form $\psi = (\psi_0, \psi_1, \psi_2)$. When UCT generates tree nodes at depths greater than D, we prune using ψ_2. As previously mentioned, we do not limit the rollouts of UCT to depth D. Rather, the rollouts proceed until terminal states, which provide a longer-term heuristic evaluation for tree nodes. However, we have confirmed experimentally that such nodes are not visited frequently enough for pruning to have a significant impact.

As previously mentioned, we learn partial policies by learning linear scoring functions. This requires us to compute state-action features $\phi(s, a)$ that capture salient attributes. We now describe the features used in each domain.

Galcon features: The features are defined over the next state (or "afterstate") produced by simulating the action in the current state. The salient attributes of the next state for launch actions include attributes of the source and destination planet like the planet size, its current population, the growth rate, and the incoming enemy threat. All real-valued features are discretized into 100 bins, using a one-hot encoding. For destination planets, the feature vector consists of the concatenation of three one-hot sub-vectors, corresponding to whether the destination planet is occupied by the current player, by the adversary, or unoccupied by either player. Note that exactly one of the three sub-vectors is non-zero for a particular destination planet. For "do nothing" actions, we have a single indicator feature. Finally, we also have a constant (bias) feature. This leads to a total feature space of size 1602, but only a small number (typically nine) are active for a given state-action pair. However, the computation of $\phi(s, a)$ is still slow due to the simulator call to obtain the next state.

Yahtzee features: The features for the current state-action pair are defined over the next state, which is sampled by calling the simulator on the current state-action pair. The state-action feature vector is specified using a sparse one-hot encoding, where each of the 13 categories has 100 features assigned to it with one additional bias feature. The feature value for each unfilled category is a measure of how "close" the dice are to achieving the maximum score for that category, discretized into 100 bins. A perfect score returns a feature value of 1, whereas a category selection that does not add to the game score returns a feature value of 0. However, "select" actions for the upper categories that achieve binary outcomes (e.g., either a "full house" is achieved or not), are encoded using binary indicator features. All "roll" actions use discretized features, encoding how close the roll action brings the dice to a desired configuration. Note that there is large variance in the state-action features due to the dice rolls. This variance may be reduced by sampling more states and averaging. However, doing so would significantly increase the computational overhead of computing $\phi(s, a)$.

Evaluation metrics: We report the search budget in real-time (e.g., 4 seconds per move) instead of the number of simulations. Each point on the anytime curve is the average of 1000 final game scores, with 95% confidence intervals. This large-scale evaluation has massive computational cost and requires access to a cluster. The use of time-based search budgets in a cluster environment adds variance, requiring additional evaluations and averaging. However, an important result of the following experiments is that the use of time to measure budgets is essential for a fair comparison of "knowledge-injected" search algorithms.

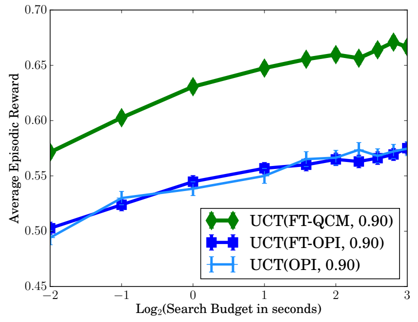
6.3 Comparing OPI, FT-OPI, and FT-QCM

We learn ranking functions using each of the three learning algorithms, as described in section 4.2. We obtain a set of partial policies by varying the pruning thresholds σ_d. Recall from section 4.2 that our implementation of rank function learning does not depend on σ_d. The learned partial policies are used to prune actions during search. Figure 5 shows the impact on search performance using UCT as the fixed base search algorithm. The first row shows the results for a large amount of pruning ($\sigma_d = 0.90$) while the second and third rows have progressively smaller pruning thresholds.

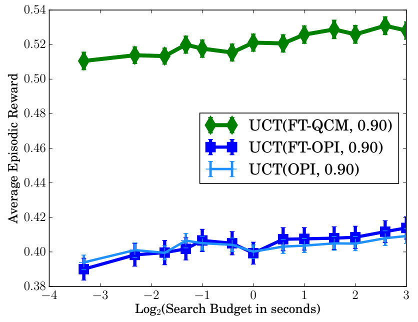
The first observation is that FT-QCM is clearly better than FT-OPI and OPI in both domains. FT-OPI and OPI are only competitive with FT-QCM in Galcon if small pruning is applied (bottom left). In Yahtzee, FT-QCM is significantly better even when the pruning is reduced to 50%.

Second, we observe that all search algorithms perform better as the search budget is increased. However, the pruning thresholds significantly impact the performance gain across the anytime curve. In Galcon, this impact is highlighted by the steep curves for $\sigma_d = 0.5$, compared to the relatively flat curves for larger pruning thresholds. In Yahtzee, although performance appears to increase very slowly, this small increase corresponds to a significant improvement in playing strength.

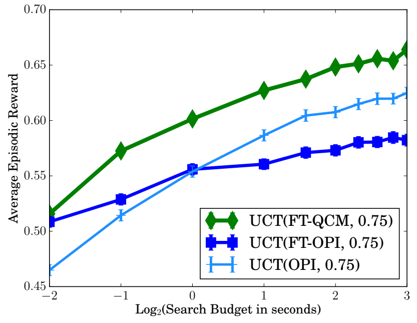
The clear superiority of FT-QCM merits further investigation. Recall that FT-QCM differs from FT-OPI only in terms of the cost function. Given the near-identical performance of FT-OPI and OPI, it seems that forward training is less relevant to performance. Therefore, we conclude that the performance improvement comes from incorporating the Q-cost instead of zero-one cost. Next, we analyze the learned partial policies to gain additional insight into the impact of Q-cost learning, compared to zero-one cost and forward training.



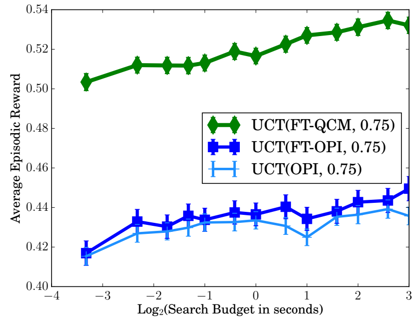
(a) Galcon, $\sigma_d = 0.90$



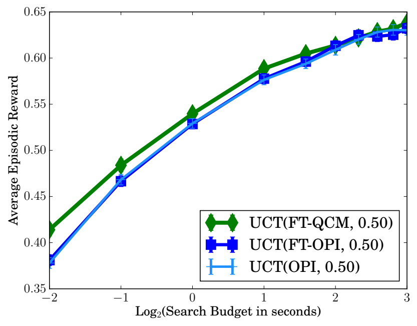
(b) Yahtzee, $\sigma_d = 0.90$



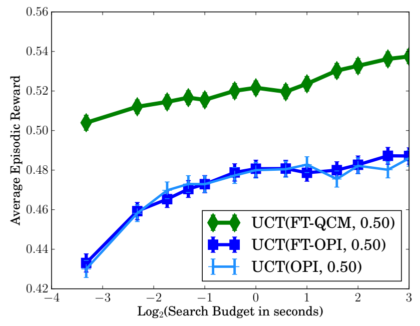
(c) Galcon, $\sigma_d = 0.75$



(d) Yahtzee, $\sigma_d = 0.75$



(e) Galcon, $\sigma_d = 0.50$



(f) Yahtzee, $\sigma_d = 0.50$

Figure 5: The search performance of UCT using partial policies learned by FT-QCM, FT-OPI, and OPI as a function of the search budget. The results for Galcon are in the first column, the results for Yahtzee in the second. Each row corresponds to a different set of pruning thresholds. The top row shows the results of pruning 90% of the actions whereas the bottom row prunes 50%. FT-QCM easily outperforms FT-OPI and OPI in five instances and is tied in the sixth. As mentioned in section 6.1, the reward (bounded by $[0, 1]$) is a linear function of the raw score, measured at the end of the game.

6.4 Relating search and regret

In order to better understand how search performance is related to the quality of the pruning policies, we analyze the learned partial policies on a held-out dataset of trajectories. In particular, for each linear partial policy, we compute its average pruning error and average regret as the level of pruning is increased from none ($\sigma_d = 0$) to maximum ($\sigma_d = 1.0$). Recall the definition of pruning error from Equation 4, which is the probability of ψ pruning away the optimal action (w.r.t. the expert policy). The definition for regret is given in Equation 3. These supervised metrics are computed by averaging over a held-out dataset of root states taken from at least one thousand trajectories. Figure 6 shows these supervised metrics on Galcon for each of the three depths. The left column shows the average regret while the right column is the average pruning error rate. The result of pruning randomly is also shown since it corresponds to pruning with random action subsets. Figure 7 shows the corresponding results for Yahtzee.

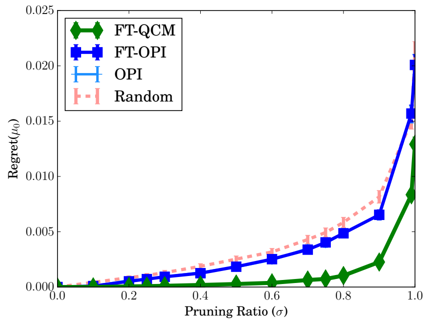
We start at the root, which is the first row in both figures. As expected, both regret and error increase as more actions are pruned. The key result here is that FT-QCM has significantly lower regret than FT-OPI and OPI for both domains. The lower regret is unsurprising given that FT-QCM uses the regret-based cost function while FT-OPI and OPI use zero-one costs. However, FT-QCM also demonstrates the least pruning error in both problems. Given that the same state distribution is used at the root, this observation merits additional analysis.

There are two main reasons why FT-OPI and OPI do not learn good policies. First, the linear policy space has very limited representational capacity. Second, the expert trajectories are noisy. The poor quality of the policy space is highlighted by the spike in the regret at maximum pruning, indicating that all three learners have difficulty imitating the expert. We also observe this at the end of training, where the final 0-1 error rate remains high. The expert's shortcomings lie in the difficulty of the problems themselves. For instance, in Galcon, very few action nodes at depths $d > 0$ receive sufficient visits. In Yahtzee, the stochasticity in dice rolls makes it very difficult to estimate action values accurately. Furthermore, in states where many actions are qualitatively similar (e.g., roll actions with nearly identical Q values), the expert's choice of action is effectively random. This adds noise to the training examples, which makes the imitation learning problem harder. However, FT-QCM's use of regret-based costs allows it to work well in Yahtzee at all depths, but only at the root in Galcon. In Galcon, the expert deteriorates rapidly away from the root policy and the training dataset becomes very noisy. Given the dominating performance of FT-QCM over OPI and FT-OPI, we focus on FT-QCM for the remainder of the experiments.

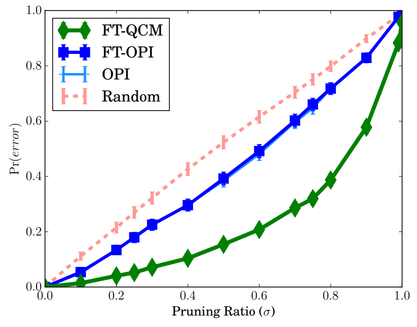
6.5 Selecting σ_d

We will now describe a simple technique for selecting σ_d. Recall that a good pruning threshold must "pay for itself". That is, in order to justify the time spent evaluating knowledge, a sufficient number of sub-optimal actions must be pruned so that the improvement produced is larger than what would have been achieved by simply searching more. At one extreme, too much pruning incurs large regret and little performance improvement with increasing search budgets. At the other extreme, if insufficient actions are pruned, search performance may be worse than that of uninformed (vanilla) search due to the additional overhead of computing state-action features.

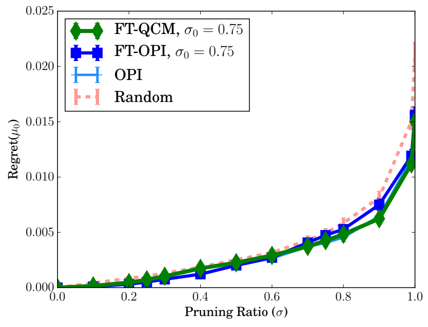
The key to selecting good values lies in the supervised metrics discussed above, where the regret and search graphs are correlated. Thus, one simple technique for selecting σ_d is to use the largest value that has sufficiently small regret. For instance, in Galcon, regret is near-zero for $\sigma_0 < 0.75$ and



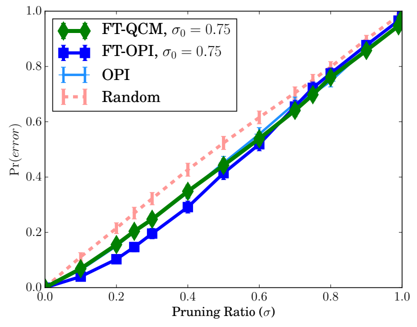
(a) Galcon: Regret at the root (d=0)



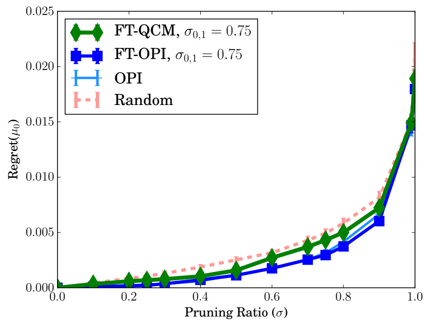
(b) Galcon: Pruning error rate at the root (d=0)



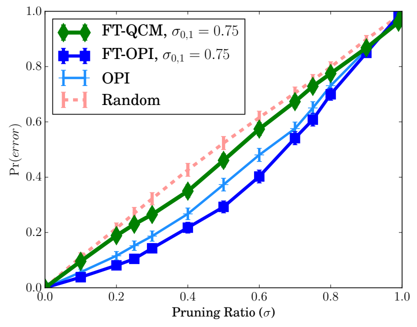
(c) Galcon: Regret at d=1



(d) Galcon: Pruning error rate at d=1

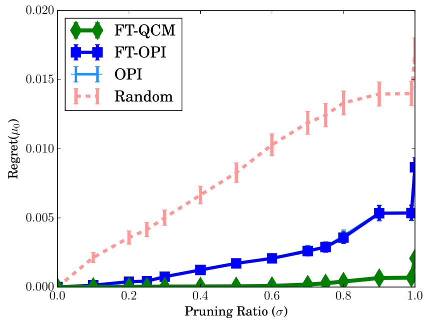


(e) Galcon: Regret at d=2

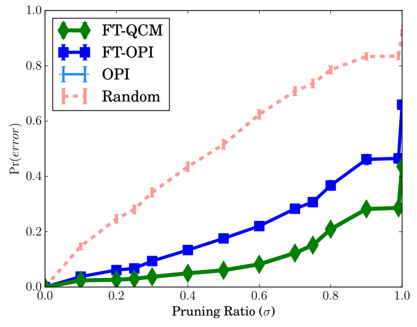


(f) Galcon: Pruning error rate at d=2

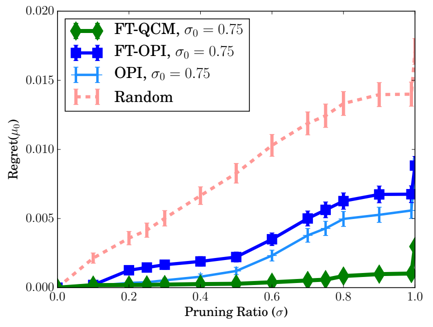
Figure 6: Average regret and pruning error for the learned partial policies ψ_d as a function of the pruning ratio in Galcon. The metrics are evaluated with respect to a held-out test set of states obtained from at least 1000 trajectories with root states sampled from μ_0. FT-QCM has the least error and regret at the root. At depths $d > 0$, performance degrades towards random due to noisy training data.



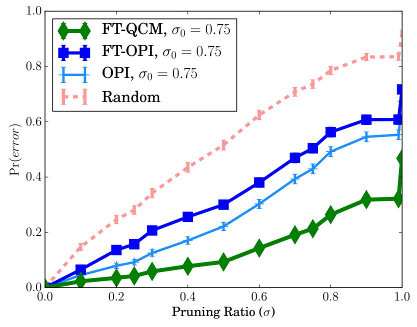
(a) Yahtzee: Regret at the root (d=0)



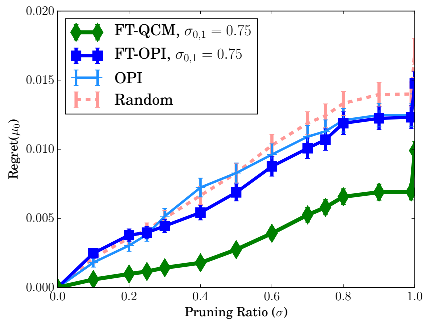
(b) Yahtzee: Pruning error rate at the root (d=0)



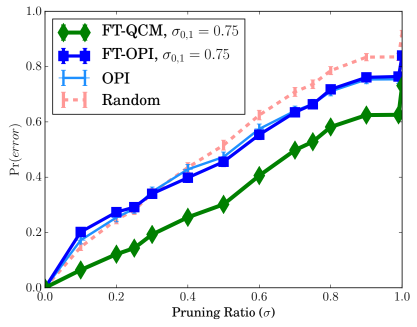
(c) Yahtzee: Regret at d=1



(d) Yahtzee: Pruning error rate at d=1



(e) Yahtzee: Regret at d=2



(f) Yahtzee: Pruning error rate at d=2

Figure 7: Average regret and pruning error for the learned partial policies ψ_d as a function of the pruning ratio in Yahtzee. The metrics are evaluated with respect to a held-out test set of states obtained from at least 1000 trajectories with root states sampled from μ_0. At depths $d > 0$, FT-QCM continues to perform better than FT-OPI and OPI due to the relatively higher quality of training data at deeper states.

increases sharply thereafter. The regret is practically zero for Yahtzee until the pruning threshold exceeds 0.5 and only seems to increase significantly around 0.75. For simplicity, we choose a constant pruning threshold and set σ_d to 0.75 for every value of d. As we will see next, using FT-QCM partial policies with $\sigma_d = 0.75$ results in large performance gains across the anytime curve. A more computationally expensive method of selecting σ_d would require the evaluation of the planning performance of a number of different pruning thresholds.

6.6 Comparing FT-QCM with baselines

We will now compare FT-QCM with other search-based and reactive agents. Most of the baselines require control knowledge in the form of either a policy or a heuristic evaluation function. Since the partial policy learned by FT-QCM at the root has the least regret, we will use it in every technique [5]. We denote this partial policy as ψ_G and its linear scoring function as $h_G(s,a) = w_G^T \phi(s,a)$.

Heuristic Bias (HB): Our first baseline is a popular technique for injecting a heuristic into the MCTS framework. HB first adds a bias to an action node's score and then slowly reduces it as the node is visited more often. This has the effect of favoring high-scoring actions early on and slowly switching to unbiased search for large budgets. This technique can be viewed as a form of progressive widening (Chaslot et al., 2007) with biased action selection and encompasses a large body of search variants. For example, one method uses the visit counts to explicitly control the branching factor at each node (Couëtoux et al., 2011a). However, none of these methods come with formal regret bounds and are agnostic to the choice of heuristic, if one is used. In our experiments, we observed that search performance is extremely sensitive to the interaction between the reward estimate, non-stationary exploration bonus, and the non-stationary bias term. In our domains, the following heuristic worked better than the other variations considered.

$$\begin{aligned} q(s,a) &= Kh_G(s,a) && \text{if } n_a = 0 \\ &= Q_{UCB}(s,a) + (1-k_n)Kh_G(s,a) && n_a > 0 \end{aligned}$$

where K is a domain-specific constant and $k_n = n(s,a)/(n(s,a)+1)$ is zero for an unvisited action and increases towards one as the action is visited more. Thus, the exploration scaling term $(1-k_n)$ is maximum (one) initially and decreases towards zero with each subsequent visit. Next, $Q_{UCB}(s,a)$ is the UCB score of an action node from Equation 6, combining the Q estimate with an exploration bonus for infrequently visited action nodes. The best values of K and the exploration constant c were obtained through a computationally expensive grid search.

Informed Rollout (IR): Another popular technique for using knowledge in UCT involves using an "informed" rollout policy. The underlying idea is that a biased rollout policy can produce more accurate state evaluations than uniform random action selection. Typically, this policy is hand-engineered using expert knowledge (e.g., MoGo (Gelly et al., 2006)), but it can also be learned offline using domain knowledge (Silver and Tesauro, 2009) or online in a domain-independent manner (Finnsson and Björnsson, 2010). Here, we use a MoGo-like strategy of selecting randomly from the partial policy ψ_G using a pruning ratio of 0.75. That is, at each state during a rollout from a leaf node, an action is randomly sampled from the top 25% of the action set with respect to the scoring function h_G. From the experiments in section 6.4, we expect this policy to have very small regret.

5. We also tried regression and RL techniques to learn an evaluation function but none of the approaches worked as well as FT-QCM.

However, "informed" rollout policies have significantly larger computational costs compared to random action selection, since each rollout requires knowledge to be evaluated a very large number of times. That is, the number of knowledge evaluations performed during policy rollouts is massive, compared to the number of knowledge evaluations performed for tree nodes. We experimented with other informed rollout policies, such as softmax as well as a complete policy that prunes all but one action. However, none of these approaches were able to perform significantly better.

UCT with Random pruning (URP): A simple yet interesting baseline is the UCT algorithm that prunes randomly. We use $\sigma_d = 0.75$ for consistency. URP avoids the cost of feature computation while simultaneously reducing the number of actions under consideration. URP is motivated by the insight obtained in the previous section, where performance at small search budgets improved with additional pruning. An important observation is that searching with random action subsets corresponds to evaluating the search performance of a random partial policy which we analyzed in Figure 6 and Figure 7.

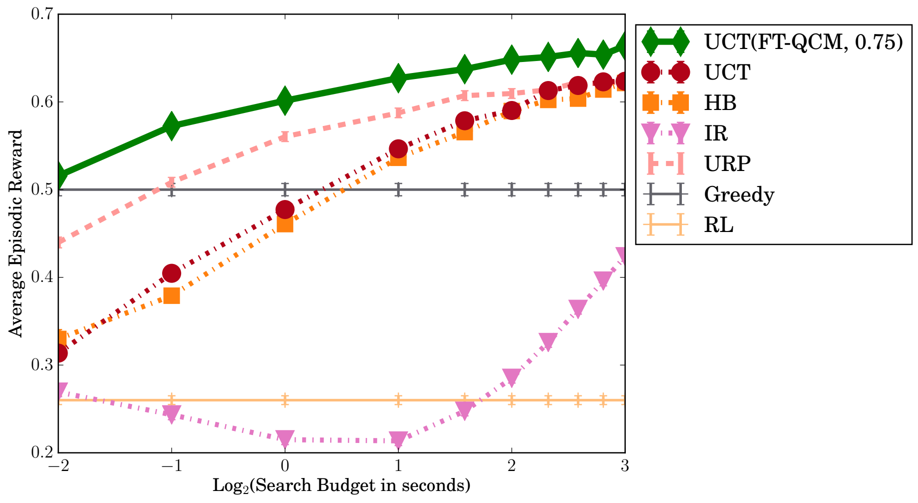
Reactive policies: Finally, we consider policies that do not search. The first, "Greedy", is simply $\arg\max_a h_G(s, a)$, acting greedily w.r.t. ψ_G. The second is a linear policy learned using the SARSA(λ) RL algorithm (Barto et al., 1995) over the same features used to learn ψ_G. We experimented with different values of λ, exploration and learning rates, and report the best settings for each domain. For Galcon, we observe that learning is negligible after 8000 games with exploration rate $\epsilon = 0.25$, decay $\lambda = 0.75$, with a constant learning rate of 0.01. For Yahtzee, which has a much faster simulator, we play 10^5 games using $\epsilon = 0.2$, $\lambda = 0.1$, and the same constant learning rate of 0.01. In both cases, RL is run until improvement becomes negligible.

Results

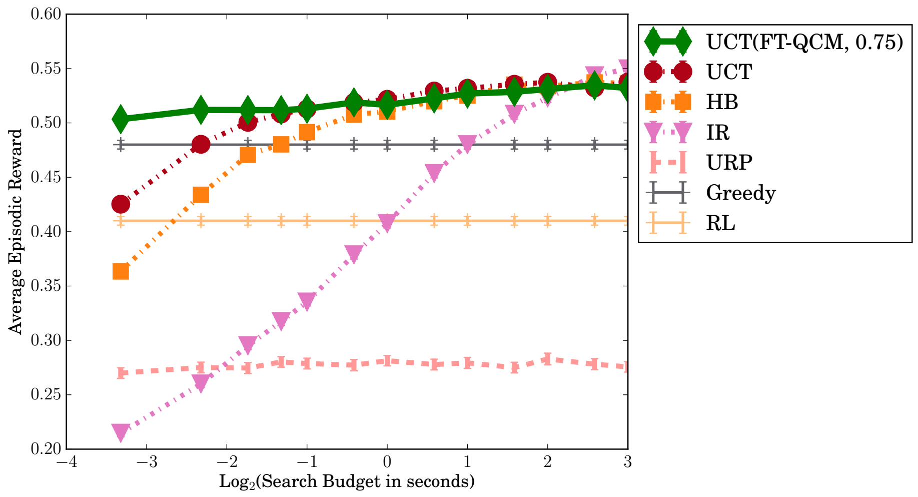
Figure 8(a) and Figure 8(b) show the results for Galcon and Yahtzee, respectively. They constitute the main experimental result of this paper. We discuss the results for each domain separately below.

Galcon: The key observation in Figure 8(a) is that FT-QCM injected into UCT significantly outperforms all other baselines at small search budgets and moderately large search budgets. The search budget is measured in real time. As the search budget increases, the FT-QCM injection continues to show slow improvement, whereas uninformed UCT fails to match performance even at eight seconds per move. We were unable to get HB and IR to perform well, which is surprising given prior reports of successful applications of these methods. IR performs particularly poorly and is worse than even the reactive policies (Greedy, SARSA). The main reason for IR's poor performance is the massive computational overhead of its "informed" rollouts. In Galcon, games (and therefore rollouts) are long which means that computing features for every legal state-action pair incurs significant computational cost. Random rollout policies are much faster in comparison and provide non-trivial value estimates. However, we observe that IR appears to be improving rapidly and may eventually outperform UCT, but only at unreasonably large search budgets (e.g., ten minutes per move).

In contrast, heuristic bias (HB) is relatively inexpensive since it must only be computed once for each state added to the tree. That is, HB has the same overhead as partial policy injection, which allows HB to perform better than IR. However, HB is unable to improve over uninformed UCT. This is surprising since prior work has described search agents that successfully leverage HB. In our experiments, we observe that HB is sensitive to the particular choice of bias decay. Getting HB to work well across the anytime curve is challenging and time-consuming. Next, we consider



(a) Galcon, FT-QCM v baselines



(b) Yahtzee, FT-QCM v baselines

Figure 8: The key empirical result of this paper shows that the search performance of UCT injected with FT-QCM significantly outperforms all other baselines at small budgets. At larger budgets, it either continues to win or achieves parity.

uninformed UCT which has zero overhead. As suspected, UCT is near-random at small time scales, getting outperformed by the variant that prunes randomly (URP). The difficulty of planning in large action spaces is highlighted by the strong performance of URP, which is second only to FT-QCM across the anytime curve. Finally, the reactive policies (Greedy, SARSA) perform reasonably well, but are incapable of utilizing larger time budgets. Greedy, in particular, appears to perform as well as UCT with a search budget of one second per move. This is much better than the policy learned by the RL algorithm. Finally, at the smallest search budget, Greedy outperforms all other baselines and is second only to FT-QCM.

Yahtzee: FT-QCM easily outperforms all baselines in this domain. However, compared to Galcon, the baselines are far more competitive at larger time scales and a number of methods achieve parity with FT-QCM. Vanilla UCT is particularly good and quickly matches FT-QCM as the search budget is increased. However, it is important to note that despite pruning 75% of the actions, FT-QCM is able to significantly outperform vanilla UCT (and other baselines) for small decision times and does not lose to vanilla UCT at larger search budgets. This result also confirms that small supervised regret (Figure 6) translates into good performance as demonstrated by the strong performance of Greedy. However, as in Galcon, both reactive policies are outperformed by search-based techniques given sufficient time.

HB and IR are uncompetitive in this domain. IR performs very poorly at first, for reasons discussed previously. However, IR eventually overtakes all methods and the graph indicates that performance may improve further, given additional time. However, HB is worse than UCT and only manages to achieve parity at large time scales. The challenges of planning in Yahtzee are clearly visible in the relatively flat anytime curves for each search variant and the high residual variance, despite averaging over a thousand games. Given the relatively small action branching factor compared to Galcon, it is notable that in Yahtzee, FT-QCM is able to perform significantly better than all competing knowledge injections and reactive policies.

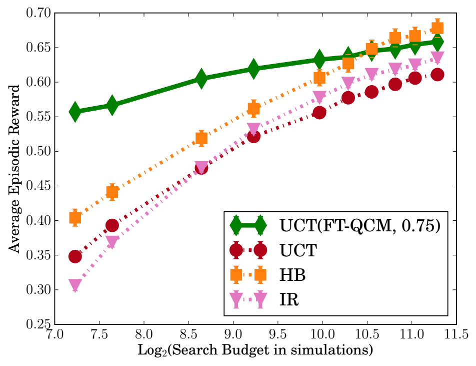
Finally, URP performs very poorly in Yahtzee, in contrast to its good performance in Galcon. The underlying reason is the relatively small percentage of good actions in a typical Yahtzee state. In Yahtzee, URP's random pruning will often result in the elimination of all reasonable actions, which decreases performance. In Galcon, this is less likely to occur since there are a much larger fraction of good actions.

6.7 The cost of knowledge

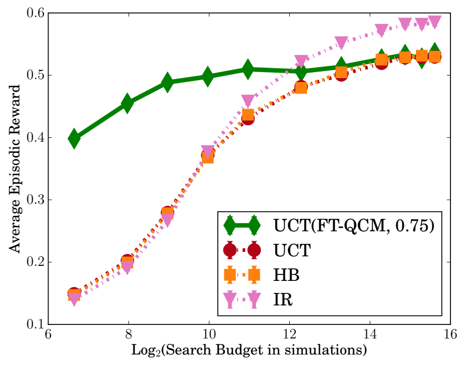
Given that many successful applications of biased initialization methods like HB and biased rollout methods like IR have exhibited improved search performance in the literature, we were surprised when HB and IR failed to improve over vanilla MCTS in our domains. Much tuning was required for HB to balance the effect of different score components. In contrast, it is relatively simple to bias rollouts and the results indicate that state-of-the-art performance is achievable with IR if one is prepared to search for very long periods of time.

The key issue seems to be the high computational cost of knowledge compared to conducting additional search. In order to evaluate the true cost of knowledge in IR and HB, we switch to measuring performance in terms of the number of simulated trajectories rather than wall-clock time. This evaluation approach ignores the time-cost of applying either the evaluation heuristic or the biased rollout policy. This reporting method has the advantage of being invariant to the architecture and implementation and is widely used in the MCTS literature. Note that we cannot re-compute these results by simply translating the time-based evaluations, since each rollout can take a variable amount of time to finish. For example, rollouts from states at the start of a game take more time compared to rollouts from near-terminal states. We therefore re-evaluate the agents, where each search is now allowed to perform a fixed number of simulations from the root. The impact of measuring search budgets using simulations instead of time is shown in Figure 9.

UCT(FT-QCM) continues to win easily at search budgets that are relatively small. However, as the search budget increases, HB and IR now demonstrate better performance. At the largest search budgets, HB outperforms UCT in Galcon and achieves parity in Yahtzee. IR eventually outperforms all other variants in Yahtzee. This improvement is noteworthy, given that IR performed very poorly in the time-based evaluation. These experiments show that using biased rollouts can improve search performance but only at exorbitant and unpredictable runtimes, varying with the particular state at the root of the tree. A broader observation is that it seems very important to correctly incorporate the cost of knowledge when comparing different injections, which has not always been the case in prior work on MCTS.



(a) Galcon, # of simulations



(b) Yahtzee, # of simulations

Figure 9: The results of measuring search performance using the number of simulations, instead of time. Doing so eliminates the computational cost of using control knowledge. IR (UCT with biased rollouts) now appears to perform much better. However, UCT with FT-QCM continues to perform the best across significant portions of the anytime curve.

7. Summary and Future Work

We have shown algorithms for offline learning of partial policies, which can be used to reduce the action branching factor in time-bounded tree search. The algorithms leverage a reduction to i.i.d. supervised learning and the expected regret is shown to be bounded. Experiments in two challenging domains show significantly improved anytime performance in Monte-Carlo Tree Search.

There are a number of promising directions to explore next. The first improvement is to the base learner for the set learning problems. Recent work has studied more principled approaches for such set learning problems via a list learning approach (Dey et al., 2013; Ross et al., 2013). While the simplicity of the ranking approach used in our work is appealing, more sophisticated approaches may yield significant performance gains in certain cases. Importantly, a major benefit of studying algorithms within a reduction framework is that we can easily incorporate any such improvements to base learners without needing to change the overall algorithm.

Another important implementation detail that requires additional exploration is the use of non-stationary partial policies. As mentioned previously, imitation learning algorithms, such as DAG-

GER (Ross et al., 2011), provide the same regret bounds as forward training using a stationary policy, with data from different distributions combined into a single dataset. It is certainly appealing to have a single policy that shares data and parameters across depth instead of the depth-indexed non-stationary partial policies considered here. However, one practical difficulty that arises in our setting is that the training data extracted from long runs of UCT gets noisier with increasing depth. Our initial attempts to mix data across depths led to poor performance. One way to control noise in the supervised datasets, irrespective of depth, might be to run a separate, long UCT from any state that is to be included in the dataset. It may be feasible to do so given sufficient computational resources. The performance of stationary partial policies compared to non-stationary partial policies is worth future consideration.

Next, we seek a method than can replace the fixed-width forward pruning with a theoretically justified progressive widening method. Doing so would allow the search to smoothly transition from reactive decision-making to full-width search at larger time scales. It would also eliminate the need to specify σ_d upfront. Ideally, the pruning level should depend on the remaining search budget. Such a time-aware search procedure has the potential to produce state-of-the-art anytime search performance. We are unaware of any method that leverages control knowledge for performing time-aware search on a per-decision basis in a principled manner.

We would like to use the learned control knowledge to improve the expert policy, which was responsible for generating the deep trees used for training. The learned partial policy is only as good as the expert, so improvements to the expert may produce large improvements in search quality, particularly at small time scales. For instance, in Yahtzee, we observe that the expert makes serious, irreversible mistakes (for example, in category assignment at the end of the rolls). Avoiding expert mistakes seems crucial to further improvement. An iterative method, alternating between learning control knowledge and using the learned information to improve the expert, is likely to produce good results. However, FT-QCM can only obtain good value estimates for unpruned actions. Thus, it is important to analyze the regret bounds and performance from training only on a subset of actions.

Finally, we are interested in exploring principled learning algorithms for other types of control knowledge that can be shown to have guarantees similar to those in this paper. It can be argued that a huge part of the success of MCTS methods is their flexibility to incorporate different forms of knowledge, which are typically engineered for each application. Automating these processes in a principled way seems to be an extremely promising direction for improving planning systems.

Acknowledgments

This work was supported by NSF grants IIS-1219258 and IIS-1619433. We also thank the editor and anonymous reviewers for their detailed feedback and insights. In particular, the original version of this work stated a much looser version of Theorem 5. The new version is based on reviewer feedback.

References

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

J Andrew Bagnell, Sham M Kakade, Jeff G Schneider, and Andrew Y Ng. Policy search by dynamic programming. In *Advances in neural information processing systems*, page None, 2003.

Radha-Krishna Balla and Alan Fern. Uct for tactical assault planning in real-time strategy games. In *International Joint Conference on Artificial Intelligence*, pages 40–45, 2009.

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

Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower Bounding Klondike Solitaire with Monte-Carlo Planning. In *International Conference on Automated Planning and Scheduling*, 2009.

Bonet Blai and Geffner Hector. Action Selection for MDPs: Anytime AO* Versus UCT. In *AAAI Conference on Artificial Intelligence*, 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. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

Vadim Bulitko and Greg Lee. Learning in Real-Time Search: A Unifying Framework. *Journal of Artificial Intelligence Research*, 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. In *Joint Conference on Information Sciences*, pages 655–661, 2007.

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

Adrien Couëtoux, Mario Milone, Matyas Brendel, Hassen Doghmen, Michele Sebag, Olivier Teytaud, et al. Continuous rapid action value estimates. In *The 3rd Asian Conference on Machine Learning*, volume 20, pages 19–31. JMLR, 2011b.

Debadeepta Dey, Tian Yu Liu, Martial Hebert, and J Andrew Bagnell. Contextual sequence prediction with application to control library optimization. *Proceedings of robotics: Science and systems VIII*, 2013.

Hilmar Finnsson. *Simulation-Based General Game Playing*. PhD thesis, Reykjavik University, 2012.

Hilmar Finnsson and Yngvi Björnsson. Learning Simulation Control in General Game-Playing Agents. In *AAAI Conference on Artificial Intelligence*, pages 954–959, 2010.

Romaric Gaudel and Michele Sebag. Feature selection as a one-player game. In *International Conference on Machine Learning*, pages 359–366, 2010.

Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *International Conference on Machine Learning*, pages 273–280, 2007.

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

Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems*, 2014.

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

Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *ICML*, volume 2, pages 267–274, 2002.

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

Thomas Keller and Malte Helmert. Trial-based heuristic tree search for finite horizon mdps. In *International Conference on Automated Planning and Scheduling*, 2013.

Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pages 282–293. 2006.

Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

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

Jean Méhat and Tristan Cazenave. Combining uct and nested monte carlo search for single-player general game playing. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4): 271–277, 2010.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Jervis Pinto and Alan Fern. Learning partial policies to speedup mdp tree search. In *Conference on Uncertainty in Artificial Intelligence*, 2014.

D. Pomerleau. ALVINN: an autonomous land vehicle in a neural network. In *Advances in Neural Information Processing Systems*, 1989.

Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *International Conference on Artificial Intelligence and Statistics*, pages 661–668, 2010.

Stephane Ross and J Andrew Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

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

Stephane Ross, Jiaji Zhou, Yisong Yue, Debadeepta Dey, and Drew Bagnell. Learning policies for contextual submodular prediction. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1364–1372, 2013.

Claude Sammut, Scott Hurst, Dana Kedzier, and Donald Michie. Learning to fly. In *International Workshop on Machine Learning*, 1992.

David Silver and Gerald Tesauro. Monte-Carlo simulation balancing. In *International Conference on Machine Learning*, pages 945–952, 2009.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Jonathan Sorg, Satinder P Singh, and Richard L Lewis. Optimal Rewards versus Leaf-Evaluation Heuristics in Planning Agents. In *AAAI Conference on Artificial Intelligence*, 2011.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.

Umar Syed and Robert E Schapire. A Reduction from Apprenticeship Learning to Classification. In *Advances in Neural Information Processing Systems*, pages 2253–2261, 2010.

Joel Veness, David Silver, Alan Blair, and William W Cohen. Bootstrapping from game tree search. In *Advances in Neural Information Processing Systems*, pages 1937–1945, 2009.

Joel Veness, Kee Siong Ng, Marcus Hutter, William Uther, and David Silver. A Monte-Carlo AIXI approximation. *Journal of Artificial Intelligence Research*, 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 Conference on Artificial Intelligence*, 2010.

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

Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *The Journal of Machine Learning Research*, 9:683–718, 2008.