Progressive Abstraction Refinement for Sparse Sampling

Jesse Hostetler and **Alan Fern** and **Thomas Dietterich**
Department of Electrical Engineering and Computer Science
Oregon State University
{hostetje, afern, tgd}@eecs.oregonstate.edu

Abstract

Monte Carlo tree search (MCTS) algorithms can encounter difficulties when solving Markov decision processes (MDPs) in which the outcomes of actions are highly stochastic. This stochastic branching can be reduced through state abstraction. In online planning with a time budget, there is a complex tradeoff between loss in performance due to overly coarse abstraction versus gain in performance from reducing the problem size. Coarse but unsound abstractions often outperform sound abstractions for practical budgets. Motivated by this, we propose a progressive abstraction refinement algorithm that refines an initially coarse abstraction during search in order to match the abstraction to the sample budget. Our experiments show that the algorithm combines the strong performance of coarse abstractions at small sample budgets with the ability to exploit larger budgets for further performance gains.

1 INTRODUCTION

When solving planning problems with a time budget, choosing the right representation is crucial. The native representation is often too detailed. It may treat many situations as distinct that are actually similar, causing the planner to spend its limited budget planning for irrelevant contingencies and fail to discover the long term consequences of actions. We should ignore details when deliberation time is limited, but given more time we should increase the level of detail to make better decisions. We consider how to design online planning algorithms that benefit from this type of progressive attention to detail.

Our approach is based on the Monte Carlo tree search (MCTS) paradigm. MCTS methods [Browne et al., 2012] select an action in a given state by constructing a lookahead search tree using a domain simulator. Because MCTS methods plan for only one state at a time, their asymptotic sample complexity is generally independent of the size of the state space. Rather, they are limited by the search depth achievable with a given time budget, which must be deep enough to estimate the quality of actions in the current state.

A full expectimax tree of depth d has of the order $O(|\mathcal{A}| \cdot B)^d$ nodes, where $|\mathcal{A}|$ is the size of the action set and B is the number of possible outcomes per action. To achieve a larger value of d, it is necessary to limit branching in the tree. One way is to reduce $|\mathcal{A}|$, an approach taken in previous works such as [Pinto and Fern, 2014]. The other approach, and the one we consider, is to reduce B.

MCTS algorithms limit B by selective sampling. Sparse sampling (SS) algorithms [Kearns et al., 2002] limit B to a constant. Trajectory sampling (TS) algorithms like UCT [Kocsis and Szepesvári, 2006] focus exploration toward actions with higher estimated values, so that B and d are small in non-optimal subtrees. In this paper, we investigate a version of sparse sampling that employs state abstraction to further reduce B.

We limit ourselves to *state aggregation* abstractions, in which the ground states are partitioned into a set of aggregate states. Classes of aggregation abstractions that guarantee bounded performance loss in tree search algorithms have been constructed based on the value function [Hostetler et al., 2014] and on bisimilarity [Jiang et al., 2014]. But while performance loss bounds guarantee asymptotic accuracy, practical time budgets may preclude running a search to convergence. Thus the accuracy and size of the abstraction interact in a complex way to determine the actual search performance. Small budgets call for coarser abstractions, while large budgets can support finer abstractions [Jiang et al., 2014].

Our observation from experience with state abstraction in MCTS has been that search with the coarsest possible abstraction — the abstraction that maps all states to a single equivalence class — often substantially outperforms search in the ground state space for moderate search budgets. Yet

the optimal policy is seldom representable in this abstract space, and there comes a point at which search with such an abstraction cannot exploit further increases in the budget.

Our main contribution is an algorithm — PARSS — that exploits the strengths of coarse abstractions while retaining the convergence and optimality guarantees of search in the ground state space. We achieve this by beginning with a coarse abstraction and refining it during search to create a more detailed abstraction, thus allowing performance to continue to improve after the point where search with the starting abstraction would have reached a plateau. We prove that with suitable implementation choices, PARSS converges to the same result and with the same worst-case sample complexity as a sparse sampling search over the ground state space. Finally, we conduct experiments demonstrating the strength of very coarse abstractions in anytime online planning and the ability of PARSS to further improve upon their performance.

2 BACKGROUND

We consider a Monte Carlo tree search (MCTS) algorithm for online planning in Markov decision processes (MDPs). A discounted MDP is a 5-tuple $M = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$, where $\mathcal{S}$ and $\mathcal{A}$ are finite sets of states and actions, $P(s'|s,a)$ is the transition function, $R(s)$ gives the instantaneous reward in s, and $\gamma \in [0,1]$ is the discount factor. Note that undiscounted problems (ie. $\gamma = 1$) pose no problem in our online planning setting.

Tree search algorithms construct sampled approximations of the *expectimax tree* over M rooted at the current state s_0. The expectimax tree itself defines an MDP over state-action histories as follows. Let $\mathcal{H}(M, s_0) = \{s_0\} \times \mathcal{A} \times \mathcal{S} \times \cdots$ be the set of state-action histories in M starting in s_0. We write $\mathcal{H}^n$ for the set of histories of length n. Given a history $h = s_0 a_1 s_1 \dots a_n s_n \in \mathcal{H}^n$, we write $s(h) \equiv s_n$ and $a(h) \equiv a_n$ for the final state and final action in the history, $p(h) \equiv s_0 a_1 s_1 \dots a_{n-1} s_{n-1}$ for the prefix of the history, and $\ell(h) \equiv n$ for the length of the history. Using this notation, we overload the P and R functions to apply to histories by defining $P(h'|h,a) = \mathbf{1}_{p(h')=h} \mathbf{1}_{a(h')=a} P(s'|s(h),a)$ and $R(h) = R(s(h))$. The *Tree MDP* induced by a general MDP M and rooted at $h_0 = s_0$ is the tuple $T(M, s_0) = \langle \mathcal{H}, \mathcal{A}, P, R, h_0, \gamma \rangle$.

A solution of a Tree MDP is a policy $\pi : \mathcal{H} \mapsto \mathcal{A}$ mapping histories to actions. The *value* of a policy is given by the value function $V^\pi(h) = R(h) + \sum_{h' \in \mathcal{H}} P(h'|h, \pi(h)) V^\pi(h')$. A policy is *optimal* if $V^\pi(h) = V^*(h)$, where $V^*(h)$ is the optimal value function, $V^*(h) = R(h) + \max_{a \in \mathcal{A}} \sum_{h' \in \mathcal{H}} P(h'|h,a) V^*(h')$. Equivalently, an optimal policy π^* is such that $\pi^*(h) = \arg\max_{a \in \mathcal{A}} Q^*(h,a)$, where $Q^*(h,a) = R(h) + \sum_{h' \in \mathcal{H}} P(h'|h,a) \max_{a' \in \mathcal{A}} Q^*(h', a')$.

2.1 SPARSE SAMPLING

Our propsed algorithm derives from the sparse sampling (SS) algorithm [Kearns et al., 2002]. SS estimates $Q^*(h_0, a)$ by constructing a sampled approximation of the Tree MDP rooted at h_0. An SS tree of depth d and width C defines a finite horizon estimate $Q^d(h,a) = R(h) + \frac{\gamma}{C} \sum_{h' \in k(h,a)} \max_{a' \in \mathcal{A}} Q^{d-1}(h', a')$ of Q^*, with terminal values $Q^0(h,a) = R(h)$. Each $k(h,a)$ is a collection of C iid samples $h' \sim P(\cdot|h,a)$, possibly containing duplicates. The greedy policy with respect to Q^d, $\pi_{\text{SS}}(h) = \arg\max_{a \in \mathcal{A}} Q^d(h,a)$, achieves bounded suboptimality with sample complexity independent of the size of the state space. Our algorithm (Section 4) is based on Forward Search Sparse Sampling (FSSS) [Walsh et al., 2010], an SS algorithm that incorporates pruning.

2.2 STATE ABSTRACTION

We consider the simplest form of state abstraction — state aggregation — in which the abstract states are the members of a partition of the ground state space. Given a Tree MDP $T = \langle \mathcal{H}, \mathcal{A}, P, R, h_0, \gamma \rangle$, a state abstraction of T is an equivalence relation χ on the set $\mathcal{H}$. The abstraction relation induces an abstract state space $\mathcal{H}/\chi$ whose members are the equivalence classes of $\mathcal{H}$ with respect to χ. We will write T/χ as shorthand for $\langle \mathcal{H}/\chi, \mathcal{A}, P, R, h_0, \gamma \rangle$. The equivalence class of h with respect to χ is denoted $[h]_\chi$, and we say that two states h and g are equivalent with respect to χ, denoted $h \simeq_\chi g$, if $[h]_\chi = [g]_\chi$. The abstraction relation for a Tree MDP must be such that $h \simeq_\chi g \Rightarrow p(h) \simeq_\chi p(g)$ to ensure that T/χ is also a Tree MDP.

An abstraction χ is *sound* if there is an optimal policy π^* over T such that $h \simeq_\chi g \Rightarrow \pi^*(h) = \pi^*(g)$ for all $h, g \in \mathcal{H}$. Classes of aggregation abstractions that are sound or that guarantee bounded performance loss in MCTS algorithms have been studied by Hostetler et al. [2014] and Jiang et al. [2014].

2.3 ABSTRACTION REFINEMENT

State equivalence abstractions form a complete lattice ordered by abstraction granularity.

Definition 1. Abstraction ψ is *finer* than χ, denoted $\psi \preceq \chi$, if $h \simeq_\psi g \Rightarrow h \simeq_\chi g$. If in addition $\psi \neq \chi$, then ψ is *strictly finer* than χ, denoted $\psi \prec \chi$.

The finest abstraction is the *bottom* or *ground* abstraction $\perp$, which maps all states to singleton sets, $[h]_\perp = \{h\}\ \forall h$. The coarsest abstraction is the *top* abstraction $\top$, which maps all ground states of the same length to the same abstract state, $[h]_\top = \mathcal{H}^{\ell(h)}\ \forall h$. Searching in the abstract problem $T/\top$ amounts to searching for the best open-loop policy in T, while searching in $T/\perp$ is equivalent to searching in the ground space.

The lattice structure of equivalence abstractions ensures that by iteratively constructing strict refinements of any initial abstraction we will eventually reach the bottom abstraction $\perp$, which is trivially sound. Furthermore, this property does not depend on how the refinements are chosen.

3 MOTIVATING EXAMPLE: THE SAVING PROBLEM

To illustrate the type of problem structure that motivates our approach to state abstraction, we created a toy problem called the Saving problem (Figure 1). The Saving problem is an episodic task in which the agent must accumulate wealth by choosing to either *save*, *invest*, or *borrow* at each time step. The *save* action always yields an immediate reward of 1. The *borrow* action takes out a "loan", which gives an immediate reward of 2 and starts a countdown timer t_b from T_b to 0. The agent cannot *borrow* again while $t_b > 0$. When t_b reaches 0, the agent receives a reward of -3, representing repaying the loan with interest. Thus the true value of *borrow* is -1, unless the episode will end before the loan is repaid. The *invest* action gives 0 immediate reward, but gives the agent the right to take the *sell* action sometime during the next T_i time steps. The *sell* action gives a reward *price*(t) if executed at time t, where each *price*$(t) \sim \text{DiscreteUniform}\{p_{\min}, p_{\max}\}$ for $p_{\min}, p_{\max} \in \mathbb{Z}$. The agent can have only one investment at a time.

We instantiate the Saving problem with $p_{\min} = -4$, $p_{\max} = 4$, $T_i = 4$, and $T_b = 4$. With these parameters, *invest* is nearly always optimal, but only if the agent takes advantage of the investment period T_i in order to sell the investment for more than $\mathbb{E}[\textit{price}] = 0$. *Borrow* is almost always the worst action, but the agent must search to a depth of at least T_b to discover its negative consequences.

Let us consider how the two extremes of state abstraction interact with the Saving problem, beginning with the ground abstraction $\perp$. To avoid falling into the *borrow* trap, the search must reach a depth of at least T_b. A full expectimax tree of depth T_b has size of the order $O((|\mathcal{A}|B)^{T_b})$, where $B = p_{\max} - p_{\min} + 1$ is the stochastic branching due to the random fluctuation of *price*. If we are restricted to a sample budget $k \ll (|\mathcal{A}|B)^{T_b}$, then the constructed tree will be very incomplete, and the agent may accidentally choose to *borrow* or not to *invest* due to sampling variance.

At the other extreme, when searching with the top abstraction $\top$, the size of the abstract tree is of the order $O(|\mathcal{A}|^{T_b})$. This tree is much smaller in practical terms, so searches will tend to give lower-variance estimates and *borrow* should be chosen less often. On the other hand, since search with $\top$ cannot discriminate between states, the value of *invest* will be estimated as $\mathbb{E}[\textit{price}] = 0$. This estimate is less than 2, which is the opportunity cost of doing *invest* and *sell* instead of doing *save* twice. Thus search with $\top$ will incorrectly choose to *save* rather than *invest*. This failure mode of open loop replanning was noted by Weinstein and Littman [2012].

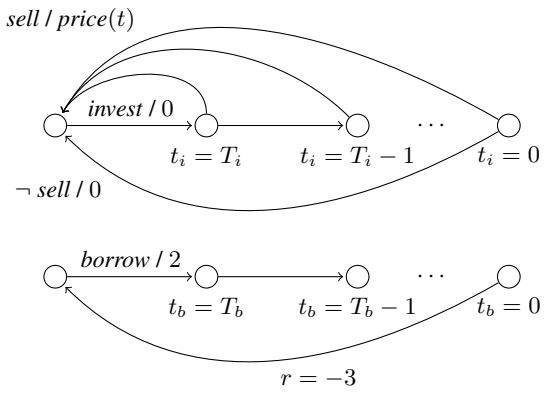

Figure 1: Schematic diagram of the *invest* and *borrow* actions in the Saving problem. Edges labeled with an action show its immediate reward.

These two extremes illustrate an important point: the appropriate abstraction depends on the sample budget. The top abstraction $\top$ is superior for small budgets. Although $\top$ is unsound in this domain, the search is operating in a much smaller state space, resulting in lower variance and less chance of incorrectly choosing *borrow*. Conversely, $\perp$ is best with a large sample budget, since $\perp$ is sound whereas coarser abstractions might not be sound. With intermediate sample budgets, it is not clear how to determine the appropriate abstraction granularity.

4 PROGRESSIVE ABSTRACTION REFINEMENT

The difficulties illustrated in the Saving problem motivate our proposed algorithm. We construct an abstract sparse sampling algorithm that begins with the coarsest abstraction $\top$ and progressively refines the abstraction during search. If the algorithm is interrupted early in the search, it gives an answer based on a coarse but inaccurate abstraction. As more samples accumulate, the abstraction is refined and the abstract search transforms smoothly into a search over the ground state space.

The algorithm is built on top of the Forward Search Sparse Sampling (FSSS) algorithm of Walsh et al. [2010]. We first describe how to modify FSSS to construct a search tree over the abstract state space induced by a fixed abstraction. We then "wrap" the abstract search in a progressive abstraction refinement procedure.

In our algorithm descriptions, we treat each history h as an object, in the sense that two histories h and g are dis-

tinct even if they describe the same sequence of states and actions. Abstract states H are collections of ground histories h, and may contain duplicates. The immediate reward for an abstract state is denoted $\mathcal{R}(H)$ and is equal to the average reward over its constituent ground states, $\mathcal{R}(H) = \frac{1}{|H|}\sum_{h\in H} R(h)$. We also assume the availability of admissible value bounds $V_{\min}$ and $V_{\max}$ such that $V_{\min} \leq V^{\pi}(h) \leq V_{\max}$ for all ground states $h \in \mathcal{H}$ and for all policies π over $\mathcal{H}$.

4.1 ABSTRACTION IN SPARSE SAMPLING

Forward Search Sparse Sampling (FSSS) [Walsh et al., 2010] is an enhancement of ordinary sparse sampling (SS) that incorporates pruning based on upper and lower bounds on the values of subtrees. It provides the same performance guarantees as SS and often does less computation.

Abstract FSSS (AFSSS; Algorithm 1) is a straightforward extension of FSSS. AFSSS constructs an FSSS tree over abstract states. The abstract tree encapsulates a tree of ground states, whose structure is defined by collections of ground successors $k(h, a)$. Each abstract *state node* is a collection $H = \{h_i\}$ of ground states. Associated with each abstract state node are an upper and a lower value bound $U(H)$ and $L(H)$[1] and a visit count $n(H)$. Each state node H such that $n(H) > 0$ has an *action node* successor Ha for each $a \in \mathcal{A}$. Action nodes have associated value bounds $U(H, a)$ and $L(H, a)$, abstraction relations $\chi(H, a)$, and abstract successor sets $K(H, a)$. The visit count $n(H, a)$ for an action node Ha is equal to the number of ground successors of Ha, $n(H, a) = \sum_{h\in H} |k(h, a)|$.

The inputs to AFSSS are an abstract FSSS tree $\mathcal{T}$, sampling width C, and maximum depth d. Like FSSS, AFSSS proceeds in a series of top-down trials that each traverse a path from the root node to a leaf state node. When extending a path, the algorithm chooses action nodes optimistically (Line 11), and chooses state nodes with the largest gap between U and L (Line 12). If the path reaches an unvisited state node (Line 9), that node is *expanded* by initializing and sampling its action node successors. The backup operation (Line 28) combines the average immediate reward over ground states with the discounted future return bounds over abstract states weighted by their empirical frequency.

When sampling an action node Ha (Line 22), the algorithm must ensure that $n(H, a) \geq C$ to satisfy the sparse sampling property. We accomplish this by sampling ground successors $h' \sim P(\cdot|h, a)$ for each ground state $h \in H$ and adding them to the ground successor collections $k(h, a)$ until $|k(h, a)| = \lceil C/|H| \rceil$ for all $h \in H$. Note that this sampling method will sometimes draw more than C samples for an abstract action node Ha. It is crucial that sampling is done in this way to allow the abstraction refinement algorithm we will build on top of AFSSS (Section 4.2) to have the same performance guarantees as FSSS.

[1] As in FSSS, these quantities bound the value estimate of the full SS tree conditioned on the samples so far. The value of an abstract state is a particular weighted average of ground values. See [Hostetler et al., 2014] for details.

Algorithm 1 Abstract Forward Search Sparse Sampling

1: **procedure** AFSSS($\mathcal{T} = \langle K, L, U, H_0, \chi\rangle$, C, d, χ_0)
2: **global** $K, L, U, H_0, \chi, C, \chi_0$
3: **while** time remains **and** not converged **do**
4: VISIT(H_0, d)
5: **procedure** VISIT(H, d)
6: **if** H is terminal **or** $d = 0$ **then**
7: $L(H) \leftarrow \mathcal{R}(H), U(H) \leftarrow \mathcal{R}(H)$
8: **else**
9: **if** $n(H) = 0$ **then** EXPAND(H, χ_0)
10: $n(H) \leftarrow n(H) + 1$
11: $a^* \leftarrow \arg\max_a U(H, a)$
12: $H^* \leftarrow \arg\max_{H'\in K(H,a^*)}[U(H') - L(H')]$
13: VISIT(H^*, $d - 1$)
14: BACKUP(H, a^*)
15: BACKUP(H)
16: **procedure** EXPAND(H)
17: **for all** $a \in \mathcal{A}$ **do**
18: $\chi(H, a) \leftarrow \chi_0(H, a)$
19: $(L(H, a), U(H, a)) \leftarrow (V_{\min}, V_{\max})$
20: SAMPLE(H, a)
21: $(L(H'), U(H')) \leftarrow (V_{\min}, V_{\max})\ \forall H' \in K(H, a)$
22: **procedure** SAMPLE(H, a)
23: **for all** $h \in H$ **do**
24: **while** $|k(h, a)| < \lceil C/|H| \rceil$ **do**
25: $h' \sim P(\cdot|h, a)$
26: $k(h, a) \leftarrow k(h, a) \cup \{h'\}$.
27: $K(H, a) \leftarrow [\bigcup_{h\in H} k(h, a)]/\chi(H, a)$
28: **procedure** BACKUP(H, a)
29: $L(H, a) \leftarrow \mathcal{R}(H) + \gamma \sum_{H'\in K(H,a)} \frac{|H'|}{n(H,a)} L(H')$
30: $U(H, a) \leftarrow \mathcal{R}(H) + \gamma \sum_{H'\in K(H,a)} \frac{|H'|}{n(H,a)} U(H')$
31: **procedure** BACKUP(H)
32: $L(H) \leftarrow \max_a L(H, a)$
33: $U(H) \leftarrow \max_a U(H, a)$

AFSSS terminates when the time budget is exceeded or the tree has converged (Line 3). The tree has converged if

$$L(H_0, a^*) \geq \max_{a\neq a^*} U(H_0, a) \quad \text{where } a^* = \arg\max_{a\in\mathcal{A}} L(H_0, a). \tag{1}$$

4.2 PROGRESSIVE ABSTRACTION REFINEMENT FOR SPARSE SAMPLING

Our proposed algorithm (Algorithm 3) begins by building an abstract FSSS tree with respect to $\top$. After building the abstract tree, it begins to refine the abstraction, and contin-

Algorithm 2 A generic abstraction refinement procedure

```
1: procedure PAR(T = ⟨K, L, U, H_0, χ⟩)
2:     Let Ha = SELECT(T)
3:     if Ha ≠ ∅ then
4:         χ(H, a) ← REFINE(χ(H, a))
5:         SPLIT(H, a, χ)
6:         UPDATETREE(H, a)
```

ues until there are no more useful refinements to perform. We call the algorithm Progressive Abstraction Refinement for Sparse Sampling (PARSS).

PARSS combines AFSSS with an instantiation of the generic refinement procedure PAR described in Algorithm 2. The PAR procedure consists of four steps. The SELECT function either returns an action node Ha whose associated abstraction relation $\chi(H, a)$ should be refined, or indicates that no refinement is to be done. The REFINE procedure is then called on the selected abstraction relation. After refinement, the tree is SPLIT recursively to respect the new abstraction. Finally, UPDATETREE re-computes the tree statistics as necessary. The implementations of SPLIT (Line 10) and UPDATETREE (Line 20) for AFSSS are straightforward. The remaining operations, SELECT and REFINE, are described in the next two sections.

After each PAR operation, PARSS calls AFSSS on the refined tree. This is necessary because refinement may have changed the value bounds of the root node such that the tree no longer satisfies the convergence criterion.

4.2.1 Implementing SELECT

To ensure soundness, the SELECT operation must eventually select all action nodes Ha such that refining Ha could change the root action choice. A selection mechanism that guarantees this is said to be *complete*.

Definition 2. A SELECT mechanism is *complete* if it returns an action node Ha, whenever such an Ha exists, such that:

1. $\chi(H, a) \succ \bot$.
2. $U(H, a) < V_{\max}$

These conditions ensure that refining $\chi(H, a)$ could potentially alter the choice of root action. Condition (2.2) excludes action nodes in which $U(H, a) = V_{\max}$, since in this case refining below Ha cannot increase $U(H, a)$ and thus cannot affect the root action choice. This situation arises when nodes have been generated by the EXPAND operation in AFSSS, but have not been visited yet.

The requirements of Definition 2 place few constraints on the order in which nodes are selected. There are several heuristic reasons to prefer refining near the root first. The most important is that actions near the root are part of more different policies than actions near the leaves. Since the value of an action node only affects the root value if that action is part of the optimal policy, refining nodes that are members of more policies makes it more likely that the refinement will affect the root value. In discounted problems, nodes at shallow depths are also less affected by discounting. These observations suggest that a breadth-first ordering is a reasonable choice.

In our experiments, we used a randomized breadth-first strategy to choose the next node to refine. Our SELECT implementation is divided into subtree selection and node selection phases. First, a subtree H_0a that is not fully refined is chosen uniformly at random. Then we find the shallowest depth d at which some descendent of H_0a satisfies the

Algorithm 3 Progressive Abstraction Refinement for SS

```
1: procedure PARSS(h_0, C, d)
2:     Let T = ⟨K, L, U, H_0, χ⟩ where
3:         K(H_0, a) = ∅   ∀a ∈ A,
4:         L(H_0) = V_min, U(H_0) = V_max,
5:         H_0 = {h_0}, χ = ⊤.
6:     AFSSS(T, C, d, ⊤)
7:     while time remains and some χ(H, a) ≻ ⊥ do
8:         PAR(T)
9:         AFSSS(T, C, d, ⊤)
10: procedure SPLIT(H, a, χ)
11:     if H is a leaf then return
12:     Let K' = ∅                          ▷ New abstract successor set
13:     for all H' ∈ K(H, a) do
14:         Let G' = H'/χ(H, a)             ▷ Refined partition
15:         for all ⟨G', a'⟩ ∈ G' × A do
16:             K' ← K' ∪ {G'}
17:             χ(G', a') ← χ(H', a')       ▷ Copy old relation
18:             SPLIT(G', a', χ)
19:     K(H, a) ← K'                        ▷ Overwrite old successor set
20: procedure UPDATETREE(H, a)
21:     for all H' ∈ K(H, a) do
22:         UPSAMPLE(H')
23:     for t from 0 to ℓ(H) do             ▷ Backup path to root
24:         for all a ∈ A do BACKUP(H, a)
25:         BACKUP(H)
26:         Let H = p(H)
27: procedure UPSAMPLE(H)
28:     if H is a leaf then
29:         L(H) ← R(H), U(H) ← R(H)
30:     else if n(H) > 0 then
31:         for all a ∈ A do
32:             SAMPLE(H, a)
33:             for all H' ∈ K(H, a) do UPSAMPLE(H')
34:             BACKUP(H, a)
35:         BACKUP(H)
```

conditions of Definition 2, and return one such descendent at depth d uniformly at random.

4.2.2 Implementing REFINE

Due to the lattice structure of partition abstractions, repeated refinements will eventually yield the ground abstraction $\perp$, provided the refinements are strict. Thus, we require that REFINE produces *strict* refinements, to guarantee that the refinement process continues to make progress.

Definition 3. REFINE is a *strict refinement function* if $\text{REFINE}(\chi) \prec \chi$.

The best choice of REFINE implementation will depend on the problem being solved. In our experiments, we tried the following two variations.

Random Refinement. The simpler approach, REFINERANDOM, first chooses the largest set H' in the partition induced by $\chi(H, a)$. It then randomly permutes the equivalence classes in $H'/\perp$ and greedily divides them into two sets of approximately equal size. This option is fast to compute but does not exploit structure in the ground state space.

Tree-based Refinement. If we have access to a set of features $\{\phi_i(h)\}$ for each state, we can take a more sophisticated approach. REFINEDT is based on an incrementally-constructed decision tree. Each abstraction relation $\chi(H, a)$ is defined by a decision tree D. The leaves of D define the members of a partition of the successors of Ha. Interior nodes are labeled with a feature i and a threshold θ. The refinement operation chooses the largest leaf node N of D and adds a new split to D dividing N into two new sets X and Y, choosing splits greedily to maximize an evaluation function $f(X, Y)$.

The evaluation function f can be designed to encourage desired properties in the partitions. For example, if χ is such that for all $H \in \mathcal{H}/\chi$, all members of H have the same optimal action and the same optimal value, then χ is sound in sparse sampling [Hostetler et al., 2014]. This condition is called a^*-irrelevance [Li et al., 2006]. We define an evaluation function that encourages a^*-irrelevance using upper bounds $u(h)$ and $u(h, a)$ for ground state values. These can be computed along with the bounds for the abstract states during the BACKUP step (Algorithm 1, Line 28).

Using these bounds on the ground states, we define the evaluation function

$$f(X, Y) = |\bar{u}(X) - \bar{u}(Y, a^*)| + |\bar{u}(Y) - \bar{u}(X, b^*)|,$$

where $\bar{u}(H) = \frac{1}{|H|}\sum_{h \in H} u(h)$, $\bar{u}(H, a) = \frac{1}{|H|}\sum_{h \in H} u(h, a)$, $a^* = \arg\max_{a \in \mathcal{A}} \bar{u}(X, a)$ and $b^* = \arg\max_{b \in \mathcal{A}} \bar{u}(Y, b)$. Splits that maximize f will tend to put ground states that have different optimal actions or different optimal values into different abstract states.

4.3 ANALYSIS OF PARSS

The PARSS algorithm can be viewed as a different way of orchestrating the sampling of a sparse tree. If run to termination, it provides the same performance guarantees with the same sample complexity as ordinary sparse sampling.

Definition 4. An abstract search tree $\mathcal{T} = \langle K, L, U, H_0 \rangle$ is an *abstract FSSS tree with respect to* χ, or an *AFSSS(χ)* tree in shorthand, if

1. For each abstract state node H, $\forall h, g \in H$, $h \simeq_\chi g$;
2. For every abstract state node H such that $n(H) > 0$, $n(H, a) \geq C$ for all $a \in \mathcal{A}$,
3. All value bounds L and U are admissible (Section 4.1),
4. $\mathcal{T}$ satisfies the AFSSS convergence criterion (1).

One can easily verify that the output of AFSSS is an abstract FSSS tree.

Proposition 1. *Consider a* PARSS *implementation where the* SELECT *and* REFINE *operations satisfy the conditions of Definitions 2 and 3. If the current search tree $\mathcal{T}$ is an abstract* FSSS *tree with respect to abstraction χ, then after one iteration of the loop in Algorithm 3, Line 7, the resulting tree $\mathcal{T}'$ is an abstract* FSSS *tree with respect to an abstraction ψ such that $\psi \prec \chi$.*

Proof. By assumption, $\text{REFINE}(\chi(H, a))$ returns a new abstraction ψ such that $\psi(H, a) \prec \chi(H, a)$, and therefore $\psi \prec \chi$. The SPLIT operation partitions the subtree Ha according to ψ, establishing condition (4.1). The UPSAMPLE loop in UPDATETREE (Line 21) adds samples and performs backups in the subtree Ha to establish (4.2) and (4.3) for the subtree. Then values are backed up from Ha to the root node (Line 23), which establishes (4.3) for the rest of the tree. Finally, the call to AFSSS (Line 9) establishes convergence (4.4). □

Proposition 2. *If* PARSS *does not exhaust its time budget, it terminates after drawing at most $(|\mathcal{A}| \cdot C)^d$ samples from the transition function P, and the resulting search tree is an abstract* FSSS *tree with respect to $\perp$.*

Proof. By Proposition 1, each iteration of the loop in Algorithm 3, Line 7 produces a strictly refined AFSSS tree. Due to the lattice structure of aggregation abstractions (Section 2.3), the abstraction relations $\chi(H, a)$ will be equal to $\perp$ for all H, a after a finite number of iterations. The tree at this point is an abstract FSSS tree with respect to $\perp$. The worst-case sample complexity occurs if all abstract nodes H in the fully-refined tree are singletons. □

Proposition 3. PARSS *achieves the same error bounds and sample complexity as ordinary sparse sampling.*

Proof. Proposition 2 establishes that PARSS yields an AFSSS($\perp$) tree $\mathcal{T}$ with the same worst-case sample complexity as SS (ie. $O((|\mathcal{A}| \cdot C)^d)$). $\mathcal{T}$ is different from a ground FSSS tree in that states that are equal in the ground representation are aggregated in $\mathcal{T}$. Because the FSSS pruning mechanism is sound [Walsh et al., 2010], $\mathcal{T}$ achieves the same error bounds as an SS tree in which identical states are aggregated. The error bounds for sparse sampling remain valid in this case [Kearns et al., 2002], thus the conclusion follows. □

5 RELATED WORK

The PARSS algorithm begins with an abstract tree search under abstraction $\top$, which as we have noted is equivalent to searching for an open loop policy. Several works have explored the use of open loop policies for value estimation. Weinstein and Littman [2012] applied this idea in continuous action MDPs, drawing on theory developed by Bubeck and Munos [2010]. Weinstein and Littman [2013] later developed a related algorithm with a different optimization mechanism and applied it to legged locomotion tasks. Hauser [2011] used forward search with open loop policies to plan in partially observable continuous spaces.

Incremental construction of state space partitions starting from the top abstraction $\top$ has been a common approach to abstraction discovery in MDPs, in algorithms such as the G algorithm [Chapman and Kaelbling, 1991], the PARTI-GAME algorithm [Moore and Atkeson, 1995], and the UTREE algorithm [McCallum, 1996]. The REFINEDT operation (Section 4.2.2) is similar to UTREE.

One closely related work is the Tree Learning Search (TLS) algorithm of Van den Broeck and Driessens [2011]. TLS contains all of the main ideas we use in PARSS, but applied in the UCT algorithm and targeted at continuous action spaces. Each state node of a TLS tree has an associated decision tree that represents a discretization of the action space. The decision trees are grown incrementally when data indicate that an existing action equivalence class leads to states with large variation in their values.

Another closely related work is that of Jiang et al. [2014], which describes a different abstraction refinement procedure for UCT. Their search proceeds in "batches" of samples. In between each batch, all of the abstraction relations are recomputed to satisfy a local approximate homomorphism criterion with respect to the sampled tree, and then the next batch of samples is drawn via search in the new abstract state space. A key difference from our approach is that their algorithm computes a particular abstraction with specified approximation bounds. With more samples, the computed abstraction becomes a better estimate of the target abstraction. In contrast, our method computes progressively finer abstractions as the sample budget increases.

Table 1: Best parameters for each algorithm. Parameter quality is measured by AUAC(w_{mag}).

Domain	Saving			Racetrack (S)			Racetrack (L)		
Algorithm	B	C	d	B	C	d	B	C	d
PARSS+DT	-	5	5	-	10	5	-	20	4
PARSS+RAND	-	5	5	-	10	5	-	20	4
TOP	-	5	5	-	20	5	-	20	4
GROUND	-	2	5	-	10	4	-	10	4
RANDOM	2	5	5	2	20	4	2	10	4
Domain	Blackjack			Advising 1			Advising 2		
PARSS+DT	-	50	2	-	5	2	-	5	2
PARSS+RAND	-	50	2	-	5	2	-	5	2
TOP	-	50	2	-	5	2	-	5	2
GROUND	-	50	2	-	2	2	-	2	2
RANDOM	2	50	2	2	5	2	2	5	2

6 EXPERIMENTS

We evaluated PARSS with both REFINEDT and REFINERANDOM refinement procedures ("PARSS+DT" and "PARSS+RAND"; Section 4.2.2) in comparison to flat FSSS ("GROUND"), AFSSS with the top abstraction ("TOP"), and AFSSS with random abstractions of different fixed branching factors ("RANDOM"). We compared the anytime performance of the algorithms with both sample budgets and time budgets.

6.1 DOMAINS

Our test domains included the Saving problem described earlier (Section 3) as well as several other benchmark domains. We chose domains that exhibit varying amounts of stochastic branching and on which we suspected that the top abstraction would not be optimal.

Racetrack. Racetrack is the classic RL domain of Barto et al. [1995]. The agent controls a car in a grid world. The state specifies the car's position and velocity, and the actions apply an acceleration $a \in \{-1, 0, 1\} \times \{-1, 0, 1\}$ to the car. The objective is to reach a goal state in as few steps as possible without crashing into a wall. We incorporate stochasticity by making both components of the acceleration action subject independently to a random "slip" with probability 0.2, which causes 0 acceleration to be applied in that direction rather than the intended amount. We used both the "small" and "large" circuits of Barto et al. [1995].

Spanish Blackjack. Spanish Blackjack is a more complicated variation of the casino game Blackjack (or "21"). In Spanish Blackjack, the player may split to a total of up to 4 hands, may double down after splitting, may re-double the same hand up to a total of three times, and may hit after doubling. There are also bonuses for making 21 in 5 cards, 6 cards, or 7 or more cards, or with either 7,7,7 or 6,7,8.

Academic Advising. The Academic Advising domain [Guerin et al., 2012] was featured at the International Planning Competition at ICAPS in 2014. The agent must take

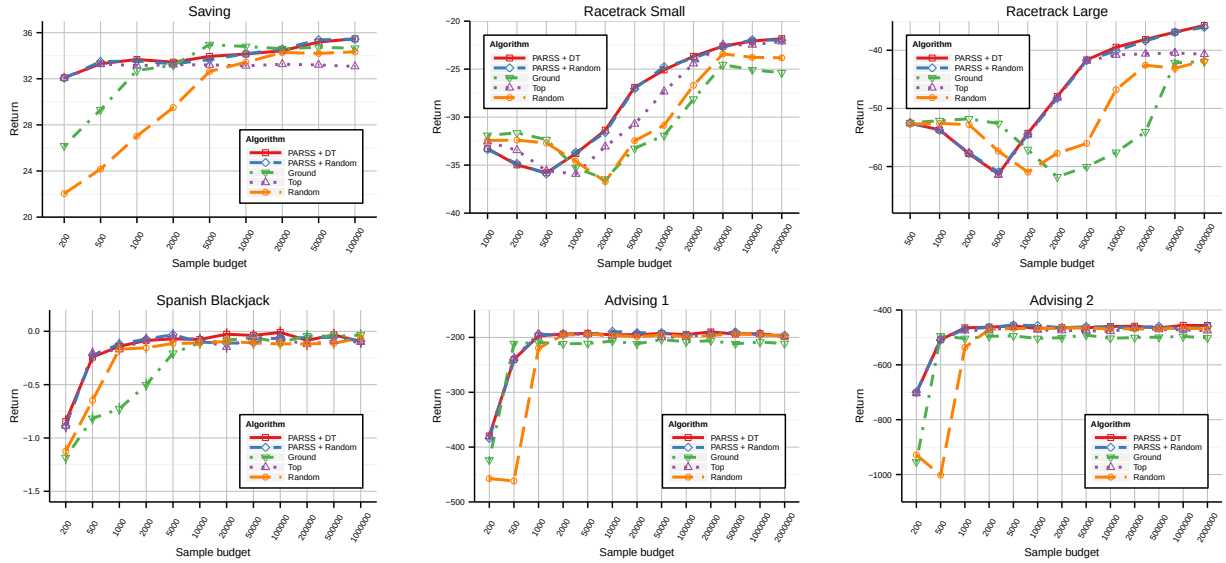

Figure 2: Performance vs. sample budget. Results are for the parameters that maximized AUAC(w_{mag}). Confidence intervals are shown, but are mostly smaller than the marker shapes except for Spanish Blackjack.

and pass all of the required courses in an academic program. The courses are linked by prerequisite relationships, and the chance of passing a course depends on how many of its prerequisites have been passed. We used MDP instances 1 and 2 from the IPC 2014.

In the IPC version of Advising, the agent can either *pass* or *fail* a course. We implemented a generalized problem that has integer grades in the range $\{0, \ldots, g\}$ to increase stochastic branching. We set $g = 4$ and set 2 as the minimum passing grade for required courses.

6.2 METHODOLOGY

We evaluated each algorithm for several combinations of the C and d parameters. The range of d spanned 3 - 4 consecutive integers for each problem and C spanned 3 - 4 consecutive values in the sequence $\{5, 10, 20, 50, 100, 200\}$. Specific ranges were chosen based on pilot experiments.

For the sample budget experiments, we restricted the algorithms to a maximum number of samples from the transition function. We evaluated sample budgets in the sequence $\{200, 500, 1000, 2000, \ldots\}$ for each parameter combination. For each budget, we measured average return over 2000 to 10000 episodes, depending on the domain.

We report results for the combinations of parameters that maximized the weighted area under the anytime curve. Given a budget sequence $\mathcal{B} = \{b_1, \ldots, b_n\}$, we calculate AUAC as

$$\text{AUAC}(w)(\mathcal{B}) = \sum_{i=2}^{n} w(b_i, b_{i-1}) \frac{\rho(b_i) + \rho(b_{i-1})}{2}, \quad (2)$$

where $\rho(b)$ is the sample average of the return of the algorithm with budget b and $w : \mathcal{B} \times \mathcal{B} \mapsto \mathbb{R}^{\geq 0}$ is a weight function. In our main experiment, we use the weight function $w_{\text{mag}}(b_i, b_{i-1}) = \log b_i - \log b_{i-1}$, which reflects a preference for good performance across orders of magnitude of budget.

Performance comparisons based on sample budgets can be misleading, since more-sophisticated algorithms do more work per sample. For those domains in which PARSS showed superior performance, we ran experiments with a time budget using the same parameters that achieved the best sample budget performance. The time budgets were measured in milliseconds and were drawn from the sequence $\{10, 16, 25, 40, 63, 100, \ldots\}$.

6.3 RESULTS

The two variations of PARSS had superior anytime performance with a sample budget in Saving and in both Racetracks (Figure 2), and were equal to TOP on the other domains. TOP outperformed GROUND in all domains except Saving but it plateaued at a suboptimal value in Saving and Racetrack Large. The pattern of performance on Saving was as expected, with TOP plateauing while both variants of PARSS continued to improve and equaled the best performance of GROUND.

In Blackjack and both Advising domains, the AUAC(w_{mag}) performance measure emphasized quick convergence, and the best C and d parameters were at the small end of their ranges (Table 6.1). TOP (and thus PARSS) converged more quickly than GROUND, and in both Advising domains also converged to a better value. PARSS did not diverge

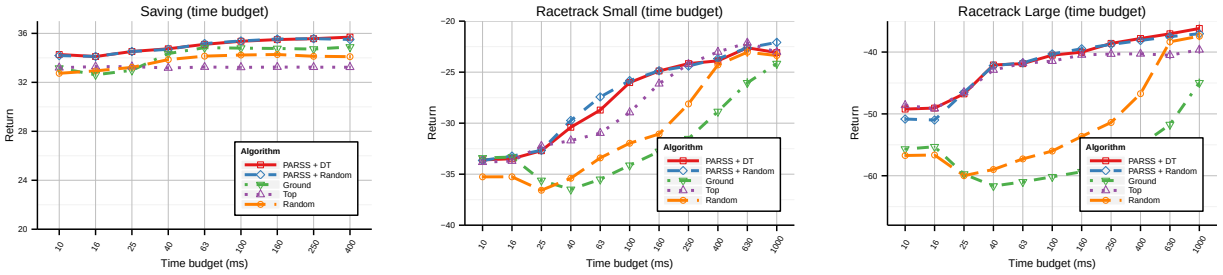

Figure 3: Performance vs. time budget. Parameter settings are the same as for the sample budget experiments.

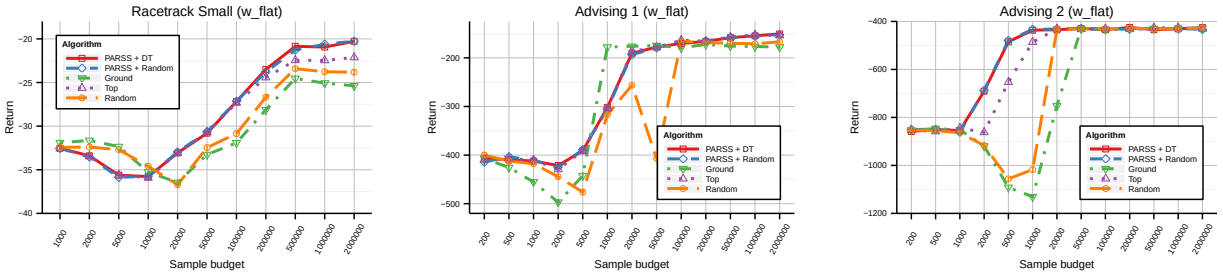

Figure 4: Selected results with best parameters according to AUAC(w_{flat}).

from TOP in these domains, indicating that abstraction refinement was not beneficial.

The RANDOM abstraction, which is a compromise between TOP and GROUND, tended to fall in between those two abstractions, although RANDOM performed poorly on Saving for small budgets.

The results with time budgets were qualitatively similar to the sample budget results (Figure 3).

To examine the sensitivity of our results to the choice of anytime performance measure, we also evaluated the algorithms with the parameters that maximized AUAC with respect to the alternative weight function $w_{\text{flat}}(b_i, b_{i-1}) = b_i - b_{i-1}$. Weighting with w_{flat} gives equal weight to all budgets in the range, and thus gives greater emphasis to large budgets compared to w_{mag}. In the domains shown (Figure 4), the different budget weighting resulted in some qualitative changes to the results, but the relative ranking of the algorithms remained the same. Note that the divergence between PARSS and TOP in Advising 2 is due to noise in parameter selection. There were no noteworthy differences in the domains not shown.

There was no difference in performance between PARSS+DT and PARSS+RAND. It appears that either REFINEDT does not find better refinements than REFINERANDOM, or else the quality of refinements does not matter much to overall performance. We expected the simpler REFINERANDOM mechanism to have an advantage under a time budget, but this was not the case.

To summarize, our main experimental findings were:

- TOP was clearly superior to GROUND overall;
- PARSS equaled the performance of TOP, or surpassed TOP through abstraction refinement;
- The two refinement mechanisms — REFINERANDOM and REFINEDT — had identical performance;
- Relative performance results with time budgets were qualitatively similar to those with sample budgets.

7 SUMMARY

We have described an extension of sparse sampling called Progressive Abstraction Refinement for Sparse Sampling (PARSS) that adapts the granularity of its state representation to the available planning budget. PARSS exploits the benefits of coarse, unsound abstractions for small budgets while transitioning smoothly to more accurate abstractions when given a larger budget. We proved that PARSS has the same sample complexity and accuracy guarantees as SS. Our experiments demonstrated that planning with the coarsest abstraction, equivalent to open loop planning, yields strong anytime performance on several benchmark domains, and confirmed that PARSS can improve upon that strong performance as budgets increase.

Acknowledgements

This research was supported by NSF grant IIS 1320943.

References

Barto, A. G., Bradtke, S. J., and Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1-2):81–138.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

Bubeck, S. and Munos, R. (2010). Open loop optimistic planning. In *Conference on Learning Theory (COLT)*.

Chapman, D. and Kaelbling, L. P. (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Guerin, J. T., Hanna, J. P., Ferland, L., Mattei, N., and Goldsmith, J. (2012). The academic advising planning domain. In *Workshop on the International Planning Competition (WS-IPC) at ICAPS*.

Hauser, K. (2011). Randomized belief-space replanning in partially-observable continuous spaces. In *Algorithmic Foundations of Robotics IX*, pages 193–209. Springer.

Hostetler, J., Fern, A., and Dietterich, T. (2014). State aggregation in Monte Carlo tree search. In *AAAI Conference on Artificial Intelligence*.

Jiang, N., Singh, S., and Lewis, R. (2014). Improving UCT planning via approximate homomorphisms. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

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

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*.

Li, L., Walsh, T. J., and Littman, M. L. (2006). Towards a unified theory of state abstraction for MDPs. In *International Symposium on Artificial Intelligence and Mathematics*.

McCallum, A. K. (1996). *Reinforcement learning with selective perception and hidden state*. PhD thesis, University of Rochester.

Moore, A. W. and Atkeson, C. G. (1995). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21(3):199–233.

Pinto, J. and Fern, A. (2014). Learning partial policies to speedup MDP tree search. In *Conference on Uncertainty in Artificial Intelligence (UAI)*.

Van den Broeck, G. and Driessens, K. (2011). Automatic discretization of actions and states in Monte-Carlo tree search. In *ECML/PKDD Workshop on Machine Learning and Data Mining in and around Games*.

Walsh, T. J., Goschin, S., and Littman, M. L. (2010). Integrating sample-based planning and model-based reinforcement learning. In *AAAI Conference on Artificial Intelligence*.

Weinstein, A. and Littman, M. L. (2012). Bandit-based planning and learning in continuous-action Markov decision processes. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

Weinstein, A. and Littman, M. L. (2013). Open-loop planning in large-scale stochastic domains. In *AAAI Conference on Artificial Intelligence*.