Improving UCT Planning via Approximate Homomorphisms

Nan Jiang
Computer Science & Eng.
University of Michigan
nanjiang@umich.edu

Satinder Singh
Computer Science & Eng.
University of Michigan
baveja@umich.edu

Richard Lewis
Department of Psychology
University of Michigan
rickl@umich.edu

ABSTRACT

In this paper we show how abstractions can help UCT's performance. Ideal abstractions are homomorphisms because they preserve optimal policies, but they rarely exist, and are computationally hard to find even when they do. We show how a combination of (i) finding *local abstractions* in the layered-DAG MDP induced by a set of UCT trajectories (rather than finding abstractions in the global MDP), and (ii) accepting *approximate homomorphisms*, leads to greater prevalence of good abstractions and makes them computationally easier to find. We propose an algorithm for finding abstractions in UCT planning and derive a lower bound on its performance. We show empirically that it improves performance on illustrative tasks, and on the game of Othello.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms

Keywords

Abstraction; UCT; Planning; Reinforcement learning

1. INTRODUCTION

State abstraction is a key ingredient in many approaches [1, 2] to applying value-function-based methods [3] to learning and planning in sequential decision-making (or reinforcement learning (RL)) problems with large or infinite state spaces. Another, quite different and more recent, approach to dealing with large state spaces in RL is the use of sample-tree-based incremental planning methods such as Sparse-Sampling [4] and the UCT [5] algorithms. At each online planning step, such methods generate a search-tree from sample trajectories and select an action greedily with respect to the estimated action-value of each action choice at the root node (the current state) computed from the tree. The remarkable result that the sample-complexity and hence computational cost for each action-selection step is *independent* of the size of state space [4, 5] accounts in large part for the popularity and success of UCT and other sample-tree-based algorithms in certain classes of domains [6, 7]. Thus, it might seem that there is no need to consider state-abstraction in UCT (we focus on UCT though our results should generalize to other sample-tree-based methods).

In this paper, we provide a number of results.

1) We show that good abstractions can help UCT's performance by improving search control, which leads to better action choices. The ideal form of abstraction is a homomorphism because the optimal policies found by planning in the abstracted problem are optimal in the unabstracted problem. But homomorphisms are global in nature, don't exist in most real problems, and even if they do exist are computationally challenging to find.

2) We present a new domain-independent algorithm for abstraction in UCT that exploits the *local layered-DAG structure* implicit in UCT's simulated trajectories to efficiently find approximate homomorphisms. By considering only the local MDP rooted at the current state, the algorithm avoids the cost of constructing abstractions in the original full MDP. It is the combination of *domain-independent* abstraction with *sample-based* planning that distinguishes our effort from other work on approximate homomorphisms in MDPs [8, 9, 10, 11].

3) Empirical tests in two illustrative domains and the game of Othello demonstrate our algorithm's effectiveness in improving UCT planning.

4) Theoretical analysis provides an upper bound for performance loss due to inaccuracy in the empirical model and approximations via abstraction.

One interesting overall insight obtained herein is that the more computationally limited UCT is, the coarser are the best-performing abstractions.

1.1 Preliminaries: MDPs and UCT

We will consider planning tasks formulated as Markov decision processes (MDPs). An MDP is a tuple $\langle S, A, T, R, \gamma \rangle$, where S is the state space and A is the action space. $T : S \times A \times S \to [0, 1]$ is the transition function, where $T(s, a, s')$ is the probability that the next state is s' given that current state is s and action a is taken, $R : S \times A \to \mathbb{R}$ is the reward function, with $R(s, a)$ being the expected immediate reward of taking action a in state s, and γ is a discount factor that determines the tradeoff between short-term and long-term rewards. A deterministic policy $\pi : S \to A$ specifies the action to take in each of the states. The value function of state s with respect to policy π is defined as

Appears in: *Alessio Lomuscio, Paul Scerri, Ana Bazzan, and Michael Huhns (eds.), Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), May 5-9, 2014, Paris, France.*

$V^\pi(s) = \mathbb{E}\big[\sum_{t=1}^{\infty} \gamma^{t-1} r_t \big| s_0 = s; \pi\big]$. Similarly, a state-action value function is defined as $Q^\pi(s,a) = \mathbb{E}\big[\sum_{t=1}^{\infty} \gamma^{t-1} r_t \big| s_0 = s, a_0 = a; \pi\big]$. The optimal policy π^* has value function and action-value function $V^* = \max_\pi V^\pi$ and $Q^* = \max_\pi Q^\pi$. Taking actions greedily with respect to Q^* yields the optimal policy π^*.

UCT. At each time step, UCT samples multiple trajectories starting from the current state to some finite depth using a generative model of the MDP, and recommends an action to execute based on the samples. When sampling, UCT decides what action to simulate by balancing exploitation and exploration. At a particular state-depth pair (or state node) (s,d), a score is computed for each action and the action with the highest score is selected. The score for action a in (s,d) is

$$Q(s,a,d) + C\sqrt{\log(n(s,d))/n(s,a,d)}. \quad (1)$$

The first component of the score, $Q(s,a,d)$, is the Monte-carlo action-value estimate from the trajectories sampled thus far of the expected discounted return obtained by taking action a at (s,d). This component reflects *exploitation.* The second component encourages *exploration* in a state (s,d) by giving a larger reward bonus to actions a taken less frequently in that state, where C is a positive parameter called the UCB scalar, $n(s,d)$ is the number of trajectories that pass through (s,d) and $n(s,a,d)$ is the number of trajectories that select action a in state s at depth d. After a prescribed number of trajectories are sampled, UCT recommends the action with the highest action-value estimate at the root node. Overall, the algorithm takes 3 parameters, the number of trajectories per step, planning depth $d_{\max}$ and UCB scalar C. Given infinite number of trajectories, the probability of recommending a suboptimal action converges to zero when the domain is episodic and $d_{\max}$ is large enough [5], otherwise there exists a truncation error bounded by γ and $d_{\max}$ [4].

2. ABSTRACTIONS HELP UCT

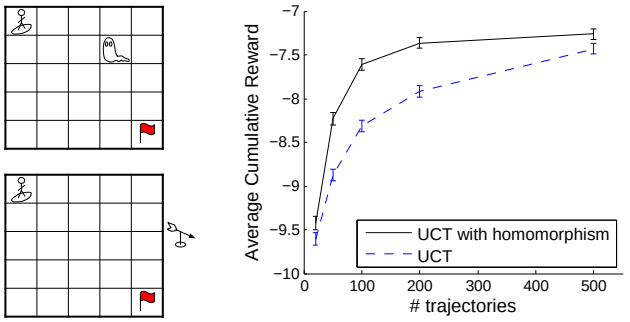
We first define perfect abstractions, i.e., global *homomorphisms*, and show that they can indeed help UCT.

Homomorphisms. A homomorphism is a perfect abstraction in the sense that the induced abstract MDP is equivalent to the original MDP for planning purposes. More formally, an MDP homomorphism h from a source MDP (unabstracted problem) $M = \langle S, A, T, R, \gamma\rangle$ to a target MDP (abstracted problem) $\tilde{M} = \langle \tilde{S}, A, \tilde{T}, \tilde{R}, \gamma\rangle$ is a surjection (mapping) from S to $\tilde{S}$ such that $\forall s \in S, a \in A, \tilde{s}' \in \tilde{S}$

$$\tilde{R}(h(s),a) = R(s,a) \text{ and } \tilde{T}(h(s),a,\tilde{s}') = \sum_{s':h(s')=\tilde{s}'} T(s,a,s').$$

The fundamental result is that the distribution over reward trajectories for any policy $\tilde{\pi}$ in $\tilde{M}$ is identical to the distribution over the reward trajectories for the corresponding *lifted* policy in M (where the lifted policy $\pi : s \mapsto \tilde{\pi}(h(s))$). As a consequence, for any optimal policy $\tilde{\pi}^*$ in $\tilde{M}$ the corresponding lifted policy is optimal in M.

Incorporating abstractions in UCT. In UCT, $Q(s,a,d)$, $n(s,d)$ and $n(s,a,d)$ are used to compute the online score that in turn helps select simulation actions in generating trajectories. Given an abstraction h that clusters unabstracted states, *whether it is a homomorphism or not*, we simply replace s with $h(s)$ when we store, retrieve and update these statistics. This is equivalent to running UCT with a simulator of the abstract MDP, while generating samples from a simulator of the unabstracted MDP.



(a) (b)

Figure 1: (a) The *Redundant-Object* domain (on top) and the *Sailing-Wind* domain (on bottom). (b) Performance comparison in *Redundant-Object* domain between UCT with and without abstraction (perfect homomorphisms).

Illustrative domain and its homomorphism. The *Redundant-object* domain is shown in the top half of Figure 1a. In a 10×10 gird world, the agent starts in top left corner and must get to the bottom right. It can move right and move down, with 0.1 probability of failure (moving in the other direction). There is a movement cost as a function of the agent's position (predetermined by randomly drawing from $U[-1,0]$ for moving down, and $U[-2,0]$ for moving right). Discount factor is 0.99. There is another randomly moving object that is visible to the agent but that has no impact on the agent. The state is the agent's position and the object's position. There is an obvious abstraction that is a (perfect) homomorphism in which the abstracted state is just the agent's position.

Results. To illustrate that global homomorphisms can help, we compare the performance of UCT with full state and UCT with abstracted state.

UCT configurations. Planning depth was fixed to 20, as an episode typically ends within 20 steps. We evaluated 20, 50, 200 and 500 trajectories. A range, $e^{\{-4,-3.6,\ldots,6\}}$, of values for the UCB exploration parameter were evaluated and the results presented are for the best choice of exploration parameters for each configuration (number of trajectories and state representation).

Figure 1b presents cumulative reward averaged over 2000 trials (with standard errors) as a function of the number of UCT trajectories. In all cases using the abstraction helped UCT. When the number of trajectories is small, the performance gap is small because UCT samples actions more than once in very few states, even with the abstraction. With a large number of trajectories, the gap is also insignificant because both algorithms are nearly-optimal.

Summary. Abstractions can help UCT's performance by changing both the exploitation component of action-values, by aggregating more trajectories into the Monte-Carlo estimates, and the exploration component of reward bonuses, by increasing visit-counts for state-depth nodes and state-depth-actions. Together these changes can lead to improved search control and thus to better action choices.

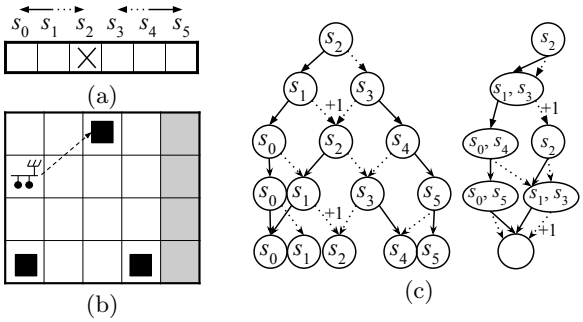


Figure 2: (a)(c) An MDP in which there is a local but no global homomorphism. (b) The *rockSample* domain.

2.1 From global & exact to local & approximate homomorphisms

There are two major challenges in exploiting global & exact homomorphisms in practice. First, non-trivial homomorphisms don't exist in most problems of interest because of the strict requirements of perfect abstraction. Second, even if non-trivial homomorphisms exist, finding the coarsest one is an NP-hard problem (see e.g., [8, 12]). The difficulty stems from the fact that aggregated states need to have the same immediate rewards as well as distributions over the next *abstract* state, which forms a recursive definition; this necessitates an iterative procedure of refining a partition over the state space until convergence, which is computationally intensive [13]. We address these challenges in three steps.

Step 1: Local Homomorphisms. At each planning step, UCT considers only the directed acyclic graph (DAG) of states reachable from the current state in a number of time steps less than or equal to the depth parameter of UCT. This induces a *local-layered MDP* (that UCT samples trajectories from) in which each node in the corresponding DAG is the pair ⟨state of the original MDP, depth in the DAG⟩. A *local homomorphism* is a homomorphism for the local-layered MDP induced from an MDP and a current state and depth. We discuss an example below.

Claim: There are MDPs in which there is no nontrivial global homomorphism, but there are local homomorphisms.

Proof by example. In the 1-d grid world shown in Figure 2a, the agent has 2 actions, one which takes a step towards the center (dotted action) and the other which takes a step away from the center (solid action); actions that would take the agent off-grid have no effect. It gets +1 reward when it moves to the location with the X symbol and gets 0 otherwise. Due to the asymmetry in the location of X, the domain has no homomorphism except the trivial one (isomorphism). The DAG for the local-layered MDP rooted at s_2 with depth 4 is shown in Figure 2c, and the DAG of the corresponding abstract local-layered MDP generated by a local homomorphism is shown on the right; the nodes clustered together level by level are shown. By inspection it can be seen that the reward for every sequence of actions is the same in both DAGs.

Since by definition all global homomorphisms are local homomorphisms, and from the example above there are local homomorphisms even when there are no global homomorphisms, we expect local homomorphisms to be a more useful target for learning abstractions for local-tree-search-based algorithms like UCT.

Claim: Computing a local homomorphism from a local-layered MDP is at worst quadratic in the number of nodes in the corresponding DAG.

Proof (constructive) by provision of a quadratic Algorithm 1 below for computing approximate local homomorphisms (exact local homomorphisms are a special case).

Thus, local homomorphisms are far easier to compute and more prevalent than global homomorphisms. Nevertheless, the requirement for exactness limits their applicability.

Step 2: Approximate Global Homomorphisms. Next we relax exactness by allowing approximation errors in a global abstraction h as follows

$$\epsilon_R = \max_{s \in S, a \in A} \left| R(s,a) - \tilde{R}(h(s),a) \right| \quad (2)$$

$$\epsilon_T = \max_{s \in S, a \in A} \sum_{\tilde{s}' \in \tilde{S}} \left| \sum_{s' \in h^{-1}(\tilde{s}')} T(s,a,s') - \tilde{T}(h(s),a,\tilde{s}') \right| \quad (3)$$

where ϵ_R is the worst-case absolute difference in expected reward, while ϵ_T is the worst-case $l1$-distance between the probability vectors of distribution over next abstract states, which falls in the range $[0, 2]$. Homomorphisms are recovered by requiring $\epsilon_R = 0$ and $\epsilon_T = 0$. The key reason to use approximate homomorphisms as the formulation of approximate abstraction is the result that the optimal policy found in $\tilde{M}$ when lifted to M incurs a loss that is upper-bounded in the worst-case by a function of ϵ_R and ϵ_T [10]. But finding a global approximate homomorphism is just as hard as finding an exact one. To address this we combine Step 1 and Step 2 next.

Step 3: A quadratic algorithm for finding Local Approximate Homomorphisms. Local approximate homomorphisms can be defined for a local-layered MDP just as for the exact homomorphisms case. Algorithm 1 (see panel; adapted from [14]) finds a local approximate homomorphism by a bottom-up procedure that aggregates nodes level-by-level in the DAG corresponding to a given local-layered MDP. It takes two input parameters (ϵ_T, ϵ_R), and clusters nodes within each level of the DAG whose approximation errors (as computed via Equations 2 and 3) are upper bounded by twice the input parameters. If the input is $(0, 0)$, perfect (possibly trivial) homomorphisms will be found. In general, the algorithm finds coarser abstractions with greater approximation parameters, ϵ_T, ϵ_R, though there will be intervals on these quantities in which the abstractions found will be the same. The quadratic in the number of nodes in the DAG computational complexity of this algorithm is clear from inspection (each node in a layer only considers nodes in the layer below and in the worst case considers each node in the layer below at most once).

3. A TRADEOFF: COARSENESS BENEFITS BUT APPROXIMATION HURTS

One concern with Algorithm 1 is that it assumes that a local-layered MDP is available at each planning step. Instead, UCT has a set of trajectories *sampled* from the local-layered MDP. We will address this practical concern in the final piece of our contribution in §4. Here, we assume access to the local-layered MDP in order to cleanly study the following tradeoff. Recall that with exact homomorphisms the theoretical guarantees of UCT are retained. With approximate homomorphisms, however, this guarantee no longer applies.

Algorithm 1 Backwards-induction algorithm for finding approximate homomorphisms in a local-layered MDP.
Input parameter: (ϵ_T, ϵ_R). Let $d(s)$ be the depth of a state and $S_d = \{s : d(s) = d\}$. The aggregation of states at depth d is represented as h_d.

Pick arbitrary s' in $S_{d_{\max}}$.
$h_{d_{\max}} \leftarrow \forall s, s \mapsto s'$.
for $d = d_{\max} - 1$ to 0 **do**
  $h_d \leftarrow$ empty map.
  for all $s \in S_d$ **do**
    if ($\exists s_1$ in the value set of h_d, s.t. $\forall a \in A$, $|R(s,a) - R(s_1,a)| \leq \epsilon_R$ and $\sum_{\tilde{s}' \in h_{d+1}(S_{d+1})} \left| \sum_{s' \in h_{d+1}^{-1}(\tilde{s}')} (T(s,a,s') - T(s_1,a,s')) \right| \leq \epsilon_T$) **then**
      add $(s \mapsto s_1)$ to h_d
    else
      add $(s \mapsto s)$ to h_d.
    end if
  end for
end for
return $h : s \mapsto h_{d(s)}(s)$

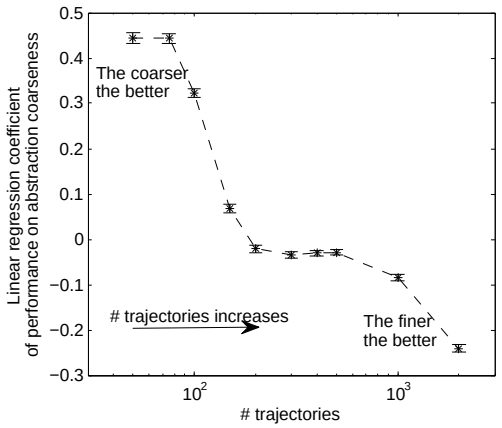


Figure 3: Relation between abstraction-coarseness and algorithm performance given different number of trajectories available to UCT.

Intuitively, UCT should benefit from coarseness of the abstraction but be harmed by the inaccuracy/approximation in the abstraction. This sets up a tradeoff that we explore in this section. Our specific hypothesis is that the fewer computational resources available to UCT (the fewer trajectories), the coarser the ideal abstraction.

Domain Description. The *Sailing-Wind* domain shown in Figure 1a (bottom) is inspired by a benchmark problem for testing variants of UCT [5]. The agent must get to the goal in the bottom right corner starting from the top left corner in the face of a stochastic wind that changes direction at every step according to a random walk (on five possible values with equal probabilities of decreasing, increasing and holding, with an initial value of 3 for all trials). The agent's two actions *move down* or *right* are deterministic. The state has a factored representation, (x, y, w), where (x, y) is the position of the agent and w is the wind direction. The reward function is the following: $R(x, y, w, a) = f(x, y, a) g_a(w)$, where $g_{down}(w) = \tan(w\pi/12)$ and $g_{right}(w) = \cot(w\pi/12)$, and f is the reward function used in the *Redundant-Object* domain. The discount factor $\gamma = 0.99$.

We derived approximate local homomorphisms of varying coarseness by using Algorithm 1 over a range of values[1] for (ϵ_T, ϵ_R). Then we evaluated these abstractions with many different computational bounds on UCT (specifically, 50, 75, 100, 150, 200, 300, 400, 500, 1000 and 2000 trajectories with the UCT exploration parameter optimized separately for each case). According to our hypothesis, when the number of trajectories is small, coarse abstractions should perform better. We tested this prediction by computing a linear regression of performance on coarseness (see footnote[2] for scalar definition) at each computational bound. The regression coefficients are plotted in Figure 3. The expected pattern is observed: the coefficients are positive at small numbers of trajectories and then decrease, finally dropping below zero as finer abstractions perform better.

4. LOCAL APPROXIMATE HOMOMORPHISMS FROM SAMPLE TRAJECTORIES

The last piece needed to make a practical abstraction algorithm for use with UCT is to drop the requirement of access to the true local-layered MDP in Algorithm 1, which means that the algorithm no longer has access to functions T and R, but only to the sample trajectories. The simple idea is to use the *empirical* local-layered MDP induced by a set of sampled trajectories as an approximation to the local MDP[3]. However, if the abstraction is done *after* UCT has finished generating its trajectories it will have no influence on the trajectories generated, and hence no effect on the action chosen (recall that the action chosen is greedy w.r.t the Monte-Carlo action-value estimates at the root node). Hence, we divide the UCT trajectory generation phase into a number of batches. Within each batch we run UCT as described above using the approximate homomorphism computed at the end of the previous batch as follows. After each batch we update the empirical layered-MDP using all the trajectories sampled thus far and recompute an approximate homomorphism, and finally we recompute all the UCT values (action-values and reward bonuses) to determine the sampling of the next batch of trajectories. The number of batches, denoted l, is an input parameter of the algorithm.

There are a couple of subtleties that must be dealt with. If a state-depth pair does not appear in the sampled trajectories in the batches thus far, but is encountered in later trajectories, it is treated as unabstracted. If a state-depth pair is "under-sampled" in that at least one action has not been tried thus far in the sampled trajectories, it is aggressively aggregated with all other similarly under-sampled states at

[1] $\epsilon_T = \{0, 0.33, \ldots, 2.0\}$ and $\epsilon_R = \{0, 1.0, \ldots, 7.0\}$.
[2] At each depth of the local layered MDP rooted at the initial state, the number of abstract states is divided by the number of primitive states and the ratio is then averaged over all depths except the bottom one. A ratio of 1 means no abstraction at all, while a ratio close to 0 indicates that the abstraction is very coarse. We use the negative of this ratio as an indicator of abstraction-coarseness.
[3] If the domain has a large or infinite branching factor, the empirical MDP either converges slowly or does not converge. However, large branching factor is known to be an issue for UCT itself and tackled by its variants [15, 16]. We focus on showing improvement relative to basic UCT, while leaving application of our ideas to variants of UCT to future work.

that depth. These methods for handling these challenging cases were determined through empirical evaluation of several alternate methods. With these modifications to Algorithm 1, the pseudo-code of our proposed algorithm is specified in Algorithm 2.

Algorithm 2 Algorithm for finding approximate homomorphisms based on sampled trajectories. Works with UCT using n trajectories per step. Input parameters: $(\epsilon_T, \epsilon_R, l)$; l is the number of batch updates of the empirical local MDP and ϵ_T, ϵ_R are the abstraction parameters.

$h \leftarrow$ identity function.
for $m = 1$ to l **do**
  Sample the next n/l trajectories in UCT with abstraction h.
  Build empirical model for the local layered MDP from the mn/l trajectories sampled so far.
  Construct abstraction h' using Algorithm 1 (modified as specified in §4) with parameter (ϵ_T, ϵ_R).
  $h \leftarrow h'$.
end for

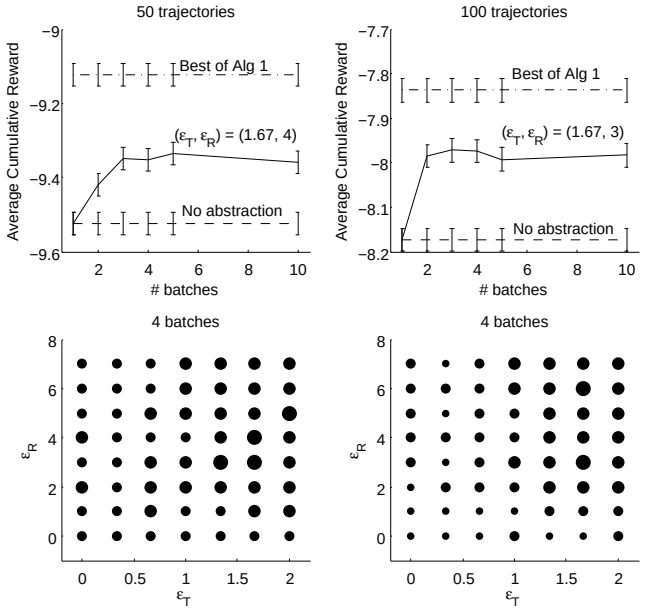


Figure 4: Performance of Algorithm 2 in *Sailing-wind* domain for 50 and 100 UCT trajectories. The top two graphs show performance as # batches (l) increases. The upper dashed line is the best performance obtained by Algorithm 1, and the lower dashed line is the performance when no abstraction is used. The fixed approximation parameters are the best picked from range indicated in the bottom two graphs. The bottom two graphs show performance as approximation parameter varies (# batches fixed to $l = 4$). The size of the $\bullet$ at a (ϵ_T, ϵ_R) position on the graph indicates the difference in performance of our Algorithm 2 from baseline (no point is worse than baseline in these two graphs) at that parameter setting. To get a sense of scale, note that the size of the $\bullet$ at $(1.67, 4)$ in the bottom left figure corresponds to the performance difference at 4 batches in the top left figure. All results are averaged over 20000 trials.

5. EMPIRICAL EVALUATION

Experiment 1: Comparing abstractions derived from local MDPs (Algorithm 1) and those derived from sampled trajectories (Algorithm 2). The objective is to see how much of the gain due to abstractions achieved by the impractical Algorithm 1 on the *Sailing-Wind domain* of §3 can be recovered by the use of the practical Algorithm 2. Figure 4 shows how our algorithm performs under different parameters for the cases of 50 and 100 UCT trajectories. In the top two graphs, the difference between the two dashed lines is the improvement achieved by the best abstraction found by Algorithm 1 ($(\epsilon_T, \epsilon_R) = (2.0, 6.0)$ for 50 trajectories and $(1.34, 4.0)$ for 100 trajectories) relative to no abstraction. Visually it is clear that Algorithm 2 (the solid line curve) recovers about half the gain of Algorithm 1. The bottom two graphs are scatter plots that show the robustness of performance against variation in the approximation parameters. See Figure4 caption for more details.

Experiment 2: Extension to POMDPs. UCT has been generalized [17] to POMDP settings by treating history as state and using a particle representation for the root state. Our Algorithm 2 also generalizes to POMDPs straightforwardly. UCT does not explicitly exploit any structure of the belief-state space, while our algorithm can potentially find and leverage such structures to improve performance in terms of abstractions. We evaluate this possibility in a benchmark POMDP problem called *rockSample* [18], shown in Figure 2b (in which there are positive-reward and negative-reward rocks and the agent can act to receive distance-dependent noisy signals about the quality of rocks; see [18] for a detailed domain description).

UCT Configuration. Planning depth is set to 20, which is sufficient considering the world's size. The number of trajectories is set to 1000, 2000 and 5000.

Results. Results are shown in Figure 5. Algorithm 2 with the best # batches parameter always outperforms UCT with no abstractions under different number of trajectories in planning (top row of graphs). Furthermore, standard UCT only has slight improvement when given more trajectories, while our algorithm's performance improves significantly. The bottom row of graphs shows that our algorithm is robust against approximation errors. Only for very large (ϵ_T, ϵ_R) is our algorithm worse than the baseline.

Parameter selection. Algorithm 2 takes three parameters: the number of batches l and the allowed approximation errors (ϵ_T, ϵ_R). How should values for these parameters be selected? For # batches, we observe the following trade-off: when the # batches is very small (such as 1 or 2), many trajectories receive no benefit from abstractions; when # batches is large (> 10), early abstractions are constructed from (inaccurate) models constructed from small numbers of trajectories, and furthermore the computational cost of finding abstractions many times is also greater. From our empirical experiments (see Figures 4 and 5), it seems that 4 is a good choice for # batches across domains, and we fix $l = 4$ for the remaining experiments. Our empirical experiments suggest that Algorithm 2 may also be robust against variation in the approximation error parameters (ϵ_T, ϵ_R) (see the scatter plots in Figure 4 and Figure 5) and so the search for good values may be easy. The result of applying the al-

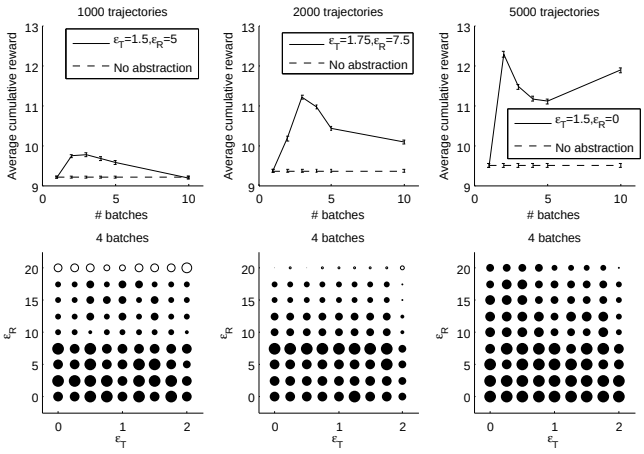


Figure 5: Performance of Algorithm 2 on the rockSample domain as #batches increases (top row) and as approximation parameters vary (bottom row). In the top graphs, the dashed line shows the performance of UCT without abstraction; the approximation parameters are the best picked from the range spanned in the bottom graphs. For the bottom graphs, a • indicates better performance than baseline (no abstraction), and a ∘ indicates worse performance; the size of the •/∘ indicates the magnitude of the performance difference. All results are averaged over 20000 trials.

gorithm to Othello, described next, is consistent with this conjecture. In future work we will consider online methods for learning good values of all three parameters.

Experiment 3: Application to *Othello*. Our final experiment is an application of Algorithm 2 to the 2-player board game of *Othello*. It has been shown that UCT-based agents can be competitive players in this game [19], and we wanted to see if we can improve their performance via abstractions. This domain is different from the previous domains in three ways. (1) The action space is inhomogeneous (different boards can have different numbers of legal moves). Only considering states with exactly the same number of actions for aggregation is too strict to find useful abstractions. Therefore, some form of action aggregation is necessary. (2) When applying UCT to board games, every trajectory has to be sampled to the end of the game because the only reward signal is a binary value indicating whether the player wins or not. Thus, with a small number of very long trajectories most states except for ones close to the root will not have all their actions explored even once. Next we propose a variation of Algorithm 2 for dealing with these issues.

Modified Algorithm. The basic modification is to truncate the tree at the depth below which there is very little search. Before aggregation, all under-sampled nodes are removed from the empirical MDP. For each remaining (s, a, d) tuple, if any of the next states (in *Othello* there is only one next state due to determinism) has been removed, (s, a, d) is treated as a terminal state that obtains a reward equal to the Monte-Carlo Q-value estimate of (s, a, d). For the truncated empirical local-layered MDP, we first perform action abstraction to combine similar actions at a state and then allow permutations when trying to match up the abstract actions of 2 states, as done in [14]. Since transitions in *Othello* are deterministic, the ϵ_T parameter no longer plays a role and so only ϵ_R need be defined for the algorithm.

Comparison to a domain-specific heuristic. We compare our algorithm to RAVE, a well-known heuristic that improves UCT performance in board games, whose efficiency has been proven in Monte-carlo *Go* [20]. RAVE generalizes value estimates among state-action pairs that share the same action based on the observation that putting a piece at a particular position usually has similar values under different boards. It is a *domain-specific* heuristic: the assumption that state-action pairs with the same action have similar values holds in board games, resource allocation domains, and some other problems with similar structures. In contrast, our algorithm generalizes value estimates among state-action pairs that are similar under the measure of approximate homomorphisms found via a *domain independent* method. But given that both approaches share the idea of generalizing value estimates, we compare our algorithm to RAVE in *Othello*. Our specific implementation follows the MC-RAVE algorithm with *handed schedule* [20] (see footnote[4] for details.)

Results. Table 1 presents the percentage of games won by the player with black pieces against the player with white pieces. Each column label is the number of UCT trajectories used by the black player vs. the white player (four combinations of 1000 and 5000). In all cases the white player uses standard unabstracted UCT. The first row of the table reports results when the black player also uses standard unabstracted UCT and thus sets the baseline for all the other rows. The rows labeled *Algorithm* 2 use the modified algorithm described above, while the rows labeled *RAVE* all use MC-RAVE. The entries in bold are statistically significantly better than the baseline entries. Thus, at least for the cases when the black player is at the same or fewer number of UCT trajectories relative to the white player, the use of our algorithm for abstraction yields about 4% to 5% improvement in performance. In the third column, the black player has many more UCT trajectories relative to the white player and hence is at a considerable advantage (baseline performance is 73% wins) which makes it harder for abstractions to improve performance. Overall, the results show that using the domain-independent abstraction algorithm can improve the performance of an already state-of-the-art algorithm, UCT, on *Othello*. On the other hand, our competitor RAVE under its best k parameter is slightly better than standard UCT, but not as good as our algorithm with the best parameter. Furthermore, as k increases to a large value, the performance is worse than standard UCT, which is expected as the algorithm relies too much on biased estimates[5].

[4]Recall that in standard UCT, the sampling policy selects actions that maximize $Q(s,a,d) + C\sqrt{\log(n_{s,d})/n_{s,a,d}}$. In RAVE, $Q(s,a)$ is replaced by a linear combination of the Monte-carlo value estimate $Q(s,a,d)$ and the All-Moves-As-First (AMAF) value $\tilde{Q}(s,a,d)$. The AMAF value is the return averaged over the trajectories that satisfy the following conditions: (1) goes through s at depth d; (2) executes action a at any point after depth d. The weight on AMAF value is denoted as $\beta(s,a,d)$, which decreases to 0 as $n_{s,a,d}$ tends to infinity so that the convergence of UCT is preserved. The *handed schedule* picks $\beta(s,a,d)$ to be $\sqrt{k/(3n_{s,d}+k)}$, where k is a positive parameter controlling the speed with which β decreases. The smaller k is, the faster β decreases and the more quickly the algorithm converges to standard UCT. We present results for $k = 1, 2, 5, 10, 100, 1000$.

[5]This phenomenon is not seen in the experiments with *Go*, where performance is maintained even for very large k,

Table 1: Results from *Othello*, comparing performance of Algorithm 2 (modified to deal with under-sampling as described in the text) to a domain-specific heuristic (RAVE) previously used in Othello and other board games. Batch parameter is fixed to $l = 4$. Table entries are winning percentage for black. The numbers in parentheses in the first row are the standard errors; they are approximately the same for the remaining rows. See text for description of the RAVE parameter k and other details.

	# black trajectories vs. # white trajectories			
Black player	1000 vs 1000	1000 vs 5000	5000 vs 1000	5000 vs 5000
UCT	45.9(0.8)	20.7(0.7)	73.0(0.7)	43.7(0.8)
Algorithm 2 ($\epsilon_R = 0.0$)	48.5(1.1)	**24.5**(0.9)	73.2(0.9)	**47.2**(1.1)
Algorithm 2 ($\epsilon_R = 0.1$)	**49.8**	22.3	73.1	**48.9**
Algorithm 2 ($\epsilon_R = 0.2$)	46.9	21.8	74.7	46.3
Algorithm 2 ($\epsilon_R = 0.5$)	42.0	19.3	73.9	**46.6**
RAVE ($k = 1$)	48.0	20.3	71.8	**46.6**
RAVE ($k = 2$)	47.2	21.7	70.8	43.8
RAVE ($k = 5$)	46.4	22.8	71.9	44.4
RAVE ($k = 10$)	45.7	19.5	71.0	44.7
RAVE ($k = 100$)	44.6	20.6	69.9	40.3
RAVE ($k = 1000$)	38.7	16.4	65.4	36.9

6. BOUNDING PERFORMANCE LOSS

Algorithm 2 builds empirical local MDPs from trajectories sampled by UCT and finds approximate homomorphisms in the empirical local MDPs. In previous sections we have shown that this can help improve performance of UCT. In this section, we provide a theoretical bound on how lossy (in sleeking actions) the constructed approximate homomorphisms could be relative to the true local MDP. The construction is lossy in two ways: (1) inaccuracy in the empirical MDP (controlled by the number of trajectories, n, sampled by UCT), and (2) approximation errors allowed in the abstractions (controlled by input parameters to the algorithm). Note that even if UCT was allowed an infinite number of trajectories, in which case the empirical local MDP would converge to the true local MDP, the approximate homomorphism based procedure would still generate loss in value. Next we define the notion of loss in value along with some notation useful for the analysis.

Notation & Objective of Analysis: For the current state of interest, let the true local layered MDP with depth $d_{\max}$ be M. In the first step that introduces value-loss, UCT samples n trajectories in M and builds an empirical MDP $\hat{M}$. In a second step that also introduces value-loss, an approximate homomorphism h maps $\hat{M}$ to an abstract MDP $\hat{M}_h$ constructed by applying Algorithm 1 to $\hat{M}$ with parameter $(\frac{\epsilon'_T}{2}, \frac{\epsilon'_R}{2})$ (hence the approximation error of the constructed abstraction is at most $(\epsilon'_T, \epsilon'_R)$) [6]. Our *analytical objective* is to bound the loss of the abstraction, i.e., to bound $\|V_M^{\pi^*} - V_M^{\pi_h^*}\|_\infty$, where V_M^π is the expected value function of policy π evaluated in MDP M, and π^* is the optimal policy in M, while π_h^* is the optimal policy in $\hat{M}_h$ lifted to true local MDP M.

Our main result, presented next, establishes that given any choice of error-parameters η_T and η_R, there is a number of trajectories, above which the loss introduced by the first of the two sources of error (inaccuracy in the empirical MDP) is bounded with high probability. Of course, the loss function also contains terms that are a function of ϵ'_T and ϵ'_R, the second of the two sources of error (approximate homomorphisms).

which shows that ASAF is a heuristic that works well for *Go* but not as well for *Othello*.

[6] We use P, $\hat{P}$ and $\hat{P}_h$ to distinguish transition probabilities of M, $\hat{M}$ and $\hat{M}_h$, and similarly for reward.

Theorem 1 (Main Result). $\forall \eta_T, \eta_R > 0, \delta \in (0,1)$,

$$\|V_M^{\pi^*} - V_M^{\pi_h^*}\|_\infty \leq \frac{2(\epsilon'_R + \eta_R)}{1-\gamma} + \frac{\gamma(R_{\max} - R_{\min})(\epsilon'_T + \eta_T)}{(1-\gamma)^2}$$

holds with probability at least $1 - \delta$ *when*

$$n > \max\{\exp^{(d_{\max})}\left[\log\left(a(KB)^{d_{\max}}/\delta\right)/b\right],\ N\}$$

and $a \stackrel{def}{=} \max\{3d_{\max},\ 6B\}$

$$b \stackrel{def}{=} \min\{2cp^2,\ 2c\eta_R^2/(R_{\max} - R_{\min})^2,\ 2c\eta_T^2/B^2\}$$

where (1) n is the number of UCT trajectories, (2) K is the number of actions under each state, (3) B is the maximal number of possible next states from a state-action pair, (4) $p \stackrel{def}{=} \min_{(s,a,s_1,d):P(s,a,s_1,d))>0} P(s,a,s_1,d))/2$*, (5)* $[R_{\min}, R_{\max}]$ *is the range of reward in $M, \hat{M}$ and $\hat{M}_h$, (6) N, c are positive constants that do not depend on the choice of η_T, η_R.*

We sketch the proof of Theorem 1 with the help of the following three lemmas:

Lemma 2. $\exists N, c > 0, \forall \eta_T, \eta_R > 0$, *when* $n > N$, $\max_{s,a,d} \sum_{s_1} \left|\hat{P}(s,a,s_1,d) - P(s,a,s_1,d)\right| \leq \eta_T$ *and* $\max_{s,a,d} |\hat{R}(s,a,d) - R(s,a,d)| \leq \eta_R$ *holds with probability at least*

$$1 - (KB)^{d_{\max}}\Big(d_{\max}\exp(-2p^2c\log^{(d_{\max})}(n)) + 2\exp(-2c\log^{(d_{\max})}(n)\eta_R^2/(R_{\max} - R_{\min})^2) + 2B\exp(-2\eta_T^2c\log^{(d_{\max})}(n)/B^2)\Big) \quad (4)$$

Lemma 2 states that for any η_R, η_T there is a number of trajectories after which the error in the empirical MDP's reward function and transition probabilities are bounded by the given parameters with high probability. Given failure probability δ, we use this bound to find satisfying n in Theorem 1.

Lemma 3. *Let the approximation errors for $h : M \mapsto \hat{M}_h$ be (ϵ_T, ϵ_R). If the conditions for Lemma 2 are satisfied, then $\epsilon_T \leq \epsilon'_T + \eta_T$ and $\epsilon_R \leq \epsilon'_R + \eta_R$.*

Lemma 3 is essentially triangle inequality for distances; the proof is elementary and hence omitted here.

Lemma 4. *[Ravindran and Barto, 2004] If the conditions for Lemma 2 are satisfied, then*

$$\|V_M^{\pi^*} - V_M^{\pi_h^*}\|_\infty \leq \frac{2\epsilon_R}{1-\gamma} + \frac{\gamma(R_{\max} - R_{\min})\epsilon_T}{(1-\gamma)^2}$$

The main result of Theorem 1 follows from Lemma 4 and so the novel part is our proof of Lemma 2 which we can only briefly sketch here because of space constraints. The proof has two steps: (1) bound the probability that $\hat{P}$ and $\hat{R}$ are not (η_T, η_R) accurate for some arbitrary (s, a, d), and (2) apply that to the whole empirical MDP by union bound. To obtained (1), we first bound $n_{s,a,d}$ (the number of visits of (s, a, d)), and then the result follows directly from Hoeffding's inequality. The bound for visit counts is given in the following lemma.

Lemma 5. $\exists N, c > 0$ *s.t.* $\forall n > N$,

$$\mathbb{P}\Big\{n_{s,a,d} \geq c \log^{(d+1)}(n)\Big\} \geq 1 - \exp(-2p^2 c \log^{(d_{\max})}(n)))^d$$

Proof. The lemma is proved by induction and builds crucially on a result of Kocsis and Szepesvari (2007; their Theorem 3) that provides a lower bound on the number of times an action is tried in a state-depth pair as a function of how many times that state-depth pair has been visited. The complete version of the proof can be found in `http://web.eecs.umich.edu/~baveja/papers/aamas-2014-theory.pdf`.

7. CONCLUSION AND FUTURE WORK

We showed how online domain-independent algorithms for discovering good state abstractions can improve the performance of UCT, a state-of-the-art sample-tree-based planning method. Our contributions include: (1) providing some insight into how abstractions can help UCT by improving search control; (2) providing a practical algorithm for automatically finding abstractions via local & approximate homomorphisms in UCT planning; (3) demonstrating that our algorithm can be effective in some illustrative domains as well as in the game of *Othello*; (4) upper bounding performance loss due to model inaccuracy and approximation in abstraction. Stronger theoretical results on the usefulness of abstractions in UCT await future work.

Acknowledgments: This work was supported by NSF grant IIS-1148668. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsors.

8. REFERENCES

[1] Balaraman Ravindran and Andrew G Barto. Model minimization in hierarchical reinforcement learning. In *5th Symposium on Abstraction, Reformulation, and Approximation*, pages 196–211, 2002.

[2] Thomas G Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[3] Richard S Sutton, David A McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, pages 1057–1063, 2000.

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

[5] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *15th European Conference on Machine Learning*, pages 282–293, 2006.

[6] Yizao Wang and Sylvain Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo go. In *IEEE Symposium Computational Intelligence and Games*, pages 175–182, 2007.

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

[8] Eyal Even-Dar and Yishay Mansour. Approximate equivalence of Markov decision processes. In *Learning Theory and Kernel Machines*, pages 581–594, 2003.

[9] Norm Ferns, Prakash Panangaden, and Doina Precup. Metrics for finite Markov decision processes. In *20th Conference on Uncertainty in Artificial Intelligence*, pages 162–169, 2004.

[10] Balaraman Ravindran and A Barto. Approximate homomorphisms: A framework for nonexact minimization in Markov decision processes. In *5th International Conference on Knowledge-Based Computer Systems*, 2004.

[11] Norm Ferns, Pablo Samuel Castro, Doina Precup, and Prakash Panangaden. Methods for computing state similarity in Markov decision processes. In *22nd Conference on Uncertainty in Artificial Intelligence*, pages 174–181, 2006.

[12] Balaraman Ravindran. *An algebraic approach to abstraction in reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 2004.

[13] Robert Givan, Thomas Dean, and Matthew Greig. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence*, 147(1):163–223, 2003.

[14] Tuomas Sandholm and Satinder Singh. Lossy stochastic game abstraction with bounds. In *13th ACM Conference on Electronic Commerce*, pages 880–897, 2012.

[15] Ronald Bjarnason, Alan Fern, and Prasad Tadepalli. Lower bounding Klondike Solitaire with Monte-Carlo planning. In *19th International Conference on Automated Planning and Scheduling*, pages 26–33, 2009.

[16] Adrien Couëtoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous upper confidence trees. In *5th International Conference on Learning and Intelligent Optimization*, pages 433–445. 2011.

[17] David Silver and Joel Veness. Monte-Carlo planning in large POMDPs. *Advances in Neural Information Processing Systems 23*, 23:2164–2172, 2010.

[18] Trey Smith and Reid Simmons. Heuristic search value iteration for POMDPs. In *20th Conference on Uncertainty in Artificial Intelligence*, pages 520–527, 2004.

[19] Philip Hingston and Martin Masek. Experiments with Monte-Carlo Othello. In *IEEE Congress on Evolutionary Computation*, pages 4059–4064, 2007.

[20] Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11):1856–1875, 2011.