Proceedings of the Twenty-Sixth International Conference on
Automated Planning and Scheduling (ICAPS 2016)

OGA-UCT: On-the-Go Abstractions in UCT

Ankit Anand,* **Ritesh Noothigattu,*** **Mausam,** and **Parag Singla**
Indian Institute of Technology, Delhi
New Delhi, India
{ankit.anand, riteshn.cs112, mausam, parags}@cse.iitd.ac.in

Abstract

Recent work has begun exploring the value of domain abstractions in Monte-Carlo Tree Search (MCTS) algorithms for probabilistic planning. These algorithms automatically aggregate symmetric search nodes (states or state-action pairs) saving valuable planning time. Existing algorithms alternate between two phases: (1) abstraction computation for computing node aggregations, and (2) modified MCTS that use aggregate nodes. We believe that these algorithms do not achieve the full potential of abstractions because of disjoint phases – e.g., it can take a while to recover from erroneous abstractions, or compute better abstractions based on newly found knowledge.

In response, we propose *On-the-Go Abstractions* (OGA), a novel approach in which abstraction computation is tightly integrated into the MCTS algorithm. We implement these on top of UCT and name the resulting algorithm OGA-UCT. It has several desirable properties, including (1) rapid use of new information in modifying existing abstractions, (2) elimination of the expensive batch abstraction computation phase, and (3) focusing abstraction computation on important part of the sampled search space. We experimentally compare OGA-UCT against ASAP-UCT, a recent state-of-the-art MDP algorithm as well as vanilla UCT algorithm. We find that OGA-UCT is robust across a suite of planning competition and other MDP domains, and obtains up to 28 % quality improvements.

Introduction

Monte Carlo Tree Search (MCTS) algorithms such as UCT (Kocsis and Szepesvári 2006) have become the de-facto standard for solving large probabilistic planning problems modeled as a Markov Decision Process (MDP). They overcome an MDP's curse of dimensionality by intelligently subsampling the tree and make effective use of available planning time. UCT-based algorithms (Keller and Eyerich 2012) have won the last two probabilistic planning competitions.

Recent work has shown that the performance of MCTS algorithms can be further improved by incorporating domain abstractions. Original MCTS algorithms explore the flat state space, which can be wasteful since many states are actually symmetric and need not be considered separately. MCTS algorithms with abstractions such as AS-UCT (Jiang, Singh, and Lewis 2014) and ASAP-UCT (Anand et al. 2015) automatically compute (approximate) symmetric nodes (states, state-action pairs) in the search tree. They aggregate such nodes into an abstract node, thus reducing the subsequent UCT planning time considerably.

Both these algorithms alternate between two phases. One phase consists of an abstraction computation routine that uses the existing UCT tree to induce groups of symmetric nodes. These nodes are aggregated to construct an abstract search tree. The second phase is the (modified) UCT algorithm, which is run as per original UCT in the beginning, but is modified to incorporate the abstractions after the abstraction routine has been run at least once.

We believe that these algorithms do not achieve the full potential of abstractions because of the two disjoint phases. Since abstractions are computed on a sampled tree, they are approximate. Erroneous abstractions computed as part of one batch of abstraction computation may get corrected only after a full phase of modified UCT – this wait could severely impact the solution quality. Moreover, while UCT prefers some states over others (due to UCB exploration rule), these algorithms treat all nodes at par while computing abstractions. This wastes valuable time on less important nodes, which likely have limited impact on further planning.

We first analyze the space of algorithmic design choices for MCTS algorithms with domain abstractions. An algorithm can be phased or incremental, abstraction computation may be done uniformly on all states or adaptively, and so on. We find that existing algorithms have complimentary strengths and weaknesses. In response, we propose *On-the-Go Abstractions* (OGA), which incorporates the best-of-all-worlds design choices.

OGA is a novel (non-phased) approach in which abstraction computation is tightly integrated into MCTS. In line with previous work, we implement these on top of UCT and name the resulting algorithm OGA-UCT. OGA-UCT has several desirable properties. First, it completely eliminates the expensive batch abstraction computation routine. OGA-UCT is *incremental* in computing abstractions, i.e., as the tree gets built it is seamlessly reduced by abstraction spot checks. Second, this allows new knowledge, either for correcting old abstractions or finding new ones, to be useful

*First two authors had equal contributions.

without a significant wait. This keeps the reduced tree as accurate as possible leading to better quality solutions. Finally, where to compute abstractions is also *adaptive* – it is guided by the UCB exploration function, thus focusing computation on the more important part of the search space.

We experimentally compare OGA-UCT[1] against ASAP-UCT (the state-of-the-art abstraction-based UCT solver) as well as the vanilla UCT algorithm. We find that across a suite of planning competition and other MDP domains, OGA-UCT performs better or at par with the best technique on each domain obtaining up to 28% solution quality improvements.

Background and Related Work

A cost-minimization finite-horizon MDP (Mausam and Kolobov 2012) can be described using a 5-tuple $(S, A, \mathcal{T}, C, H)$, where S denotes the set of states, A is the set of actions, $\mathcal{T} : S \times A \times S \rightarrow [0, 1]$ is the transition function specifying the probability of reaching a state $s' \in S$ after applying an action $a \in A$ in state $s \in S$, $C : S \times A \rightarrow \mathcal{R}$ specifies the cost or reward of taking an action a in a state s and H is the total number of steps to execute.

The solution to an MDP is a policy $\pi^* : S \times D \rightarrow A$ (D is set of decision points $\in [0, H)$) specifying the action to be taken in each state at a time point $d \in D$. The objective is to minimize the long term expected cost. The minimum expected cost (value) starting in state s at time d ($V^*(s, d)$) can be calculated recursively using the Bellman equation (Bellman 1957): $V^*(s, d) = min_{a \in A}[C(s, a) + \sum_{s' \in S} V^*(s', d + 1) \cdot \mathcal{T}(s, a, s')]$. Similarly, the Q-value for a state-action pair can be calculated as $Q^*((s, d), a) = C(s, a) + \sum_{s' \in S} V^*(s', d + 1) \cdot \mathcal{T}(s, a, s')$. The classical approach to solve an MDP is to use dynamic programming based methods such as value iteration (Bellman 1957) and policy iteration (Howard 1960).

One issue with the classical approaches to solve an MDP is that they need to enumerate the entire state space and hence, don't scale well when states and action spaces become exponential. We combine two ways to reduce this blowup – tree sampling, and domain abstractions.

Monte Carlo Tree Search (MCTS)

MCTS algorithms sample parts of the MDP search tree instead of building a complete one. At any given point, the MCTS tree can be viewed as alternating layers of state and state-action pair nodes. The most popular MCTS algorithm is UCT (Upper Confidence bounds on Trees) (Kocsis and Szepesvári 2006). It cleverly balances the exploration-exploitation trade-off while building the search tree. Starting from the root of the tree, a trajectory is iteratively sampled by selecting an action in the current state (s, d) based on the UCB rule: $argmin_{a \in A}[Q(s, a, d) - K \cdot \sqrt{\frac{log(n(s,d))}{n(s,a,d)}}]$. Here, $n(s, d)$ is the number of trajectories that pass through the state s at depth d. Similarly, $n(s, a, d)$ is the number of trajectories taking action a in state s at depth d. $K > 0$ is a constant balancing exploration and exploitation. The sampling process continues until we encounter a state not already in the tree. This newly discovered node is added to the tree as a leaf and a random rollout is performed giving an estimate of Q-value at the leaf node. This Q-value is then backed up all the way to the root. As we sample more and more trajectories, we obtain better Q-estimates at the root. Once a pre-decided number of trajectories has been sampled, UCT takes the best action at the root based on the current estimate. The tree construction process is then repeated starting at the next state as the root.

[1] https://github.com/dair-iitd/oga-uct

Abstractions and MCTS

Domain abstractions have been investigated as a method to deal with large state and action spaces (Givan, Dean, and Greig 2003; Ravindran 2004; Li, Walsh, and Littman 2006). These approaches try to deduce sets of symmetric states and aggregate them into super-states thereby reducing computation. Their early application was over traditional MDP algorithms like value iteration. Due to the success of MCTS based methods for planning, there have been some recent attempts to incorporate abstractions in MCTS (Hostetler, Fern, and Dietterich 2014; Jiang, Singh, and Lewis 2014; Anand et al. 2015). The key idea in these approaches is to use the abstractions over the sampled search tree rather than constructing them over the entire search space which can be exponentially large.

Hostetler *et al.* (2014) provide a theoretical framework for defining abstractions over the trajectories in an MCTS framework. But they do not give an algorithm to compute the abstractions. Jiang *et al.* (2014) provide the first approach for computing abstractions in an MCTS search tree and use them to provide a better estimate with the same number of samples. Their AS-UCT algorithm maintains a set of abstract states at each depth. The tree is traversed over the original nodes as in UCT. After a rollout has been performed, it is used to estimate the Q-value of all the states which fall under the same abstraction as the leaf. Similarly, during the backup phase, the update is performed over all the states which are part of the same abstract state.

Jiang's work is advanced in ASAP-UCT (Anand et al. 2015), which introduces abstractions of State-Action Pairs (SAPs) in addition to the traditionally used Abstraction of States (AS). ASAP-UCT proposes a definition of abstraction of state and state-action pairs in an MCTS tree in a recursive manner. Let $\mathcal{X}$ denote the set of abstract states at depth d. Let $\mu^d_{\mathcal{E}} : S \rightarrow \mathcal{X}$ be the state abstraction function at depth d, which takes a state $s \in S$ to its abstract state $\mu^d_{\mathcal{E}}(s) \in \mathcal{X}$. Similarly, let $\mathcal{U}$ denote the set of abstract state-action pairs. $\mu^d_{\mathcal{H}} : S \times A \rightarrow \mathcal{U}$ be the SAP abstraction function at depth d, which takes a state-action pair $(s, a) \in S \times A$ to the abstract state-action pair $\mu^d_{\mathcal{H}}(s) \in \mathcal{U}$. Then, we have the following relationship.

State Abstractions: Given the SAP abstraction function $\mu^d_{\mathcal{H}}$, two states $s, s' \in S$ have the same abstraction i.e $\mu^d_{\mathcal{E}}(s) = \mu^d_{\mathcal{E}}(s')$ iff for each action $a \in A$ applicable in s, there exists an action $a' \in A$ applicable in s' such that

Design choice	AS-UCT	ASAP-UCT	PARSS	OGA-UCT
batch vs. incremental	batch	batch	incremental	incremental
uniform vs. adaptive	uniform	uniform	adaptive	adaptive
progressive vs. split-merge	split-merge	split-merge	progessive	split-merge
unit of abstraction: states or state-action pairs (SAP)	state	SAP	state	SAP
convergence to flat or aggregate nodes	aggregate	aggregate	flat	aggregate

Table 1: Properties and design choices for MCTS algorithms computing abstractions

$\mu^d_{\mathcal{H}}(s,a) = \mu^d_{\mathcal{H}}(s',a')$ and vice versa.

SAP Abstractions: Similarly, given the state abstraction function $\mu^{d+1}_{\mathcal{E}}$, two state-action pairs (s,a) and (s',a') have the same abstraction i.e $\mu^d_{\mathcal{H}}(s,a) = \mu^d_{\mathcal{H}}(s',a')$ iff 1) $C(s,a) = C(s',a')$ 2) $\forall x \in \mathcal{X} : \sum_{s'' \in S} \mathbb{1}[\mu^{d+1}_{\mathcal{E}}(s'') = x] \cdot \mathcal{T}(s,a,s'') = \sum_{s'' \in S} \mathbb{1}[\mu^{d+1}_{\mathcal{E}}(s'') = x] \cdot \mathcal{T}(s',a',s'')$ i.e the sum of transition probabilities to same abstract class at next level is equal. Here, $\mathbb{1}$ denotes the indicator function.

Anand et al (2015) illustrate the effectiveness of using SAP abstractions in addition to the state abstractions over a variety of domains. Both AS-UCT and ASAP-UCT are batch algorithms where abstractions are computed after every few rollouts in the UCT tree. These abstractions are subsequently used during all backups until the computation of next round of abstractions. Both these approaches suffer from relatively high abstraction computation time, which leads to their poor performance in some domains compared to UCT even in presence of symmetries.

There is also recent work on improving exploration in UCT when rewards are sparse using local manifold learning (Srinivasan, Talvitie, and Bowling 2015). They improve exploration by generalizing the rollout values across all nearest neighbor states based on a distance metric determined by manifold learning of state space. Another recent work (Jiang, Kulesza, and Singh 2015) investigates selecting an appropriate level of abstraction among candidate abstractions in a model-based reinforcement learning framework.

Finally, Progressive Abstraction Refinement for Sparse Sampling (PARSS) (Hostetler, Fern, and Dietterich 2015) applies abstractions to sparse sampling framework (Kearns, Mansour, and Ng 2002). PARSS begins from a fully abstract tree, and grows it by refining the abstract nodes. The refinement procedure is somewhat ad hoc in that it splits an abstract node into two nodes arbitrarily and does not use any domain information whatsoever. It is also not clear how PARSS scales with increase in depth. Its strength is its ability to find better solutions when planning time is extremely small due to heavy initial abstraction. Moreover, it is an incremental algorithm and computes abstractions in a non-batch manner. Our 'On-the-Go' abstractions are inspired by good properties of both PARSS and ASAP-UCT, which we describe in the next section.

Design Choices for Abstraction Algorithms

Abstraction computation and tree construction mutually depend on each other. If the tree is complete, the abstractions computed will be accurate; if more abstractions are known, a more reduced tree can be built saving further computation. This interplay between MCTS and abstraction computation is relatively novel and admits several important algorithmic design choices. We briefly describe these choices and place existing algorithms in this context. We find that ASAP-UCT and PARSS have complimentary strengths and weaknesses; our proposal OGA-UCT combines the best of both worlds. Table 1 summarizes this analysis.

Batch *vs.* Incremental Computation: A key design decision is "when to initiate the abstraction computation routine"? AS-UCT and ASAP-UCT are batch-style algorithms that clearly demarcate the tree construction phase from the abstraction computation phase. Abstraction computation is treated as an independent procedure that is called periodically at specific time points. Moreover, they throw away previous abstractions, and recompute them from scratch.

An alternative is an *incremental algorithm* that tightly couples abstraction computation with tree construction. For example, PARSS adds nodes to its tree *via* abstraction refinement. It only changes abstractions of a few nodes locally and does not compute everything from scratch.

We point out that batch-style algorithms are expensive and also suffer from *stale* abstractions – it can take a while to recover from erroneous abstractions (or good abstractions missed). This can potentially lead to worse solutions because one state's Q-value may be erroneously and repeated transferred to another state. Incremental algorithms have the danger of spending too much time computing abstractions and little time using them in planning.

Uniform *vs.* Adaptive Abstractions: The success of MCTS is, in part, due to adaptive sampling, which zooms in on important parts of search tree (e.g., using UCB formula). The same principle is applicable to choosing the subset of nodes on which to attempt abstraction computation. AS-UCT and ASAP-UCT treat all search nodes in the tree equally, wasting time on nodes that might be rarely visited. PARSS performs an explicit node selection and can incorporate a measure of importance in choosing nodes for refinement.

Progressive Refinement *vs.* Split-Merge: Abstraction computation may be monotonic. For example, it may start with the flattest nodes and progressively merge nodes to create aggregate super-nodes. Or it may start with the coarsest supernodes and progressively split those to create finer refinements. The former approach will create increasingly reduced trees whereas the latter will create increasingly finer trees. A very different solution is to allow the algorithm to split or merge as necessary based on new information. We call these *Split-Merge* abstractions. A newly added node or edge in the tree may identify that two previously abstracted

nodes were, in fact, not symmetric and should be split, or that two nodes now appear symmetric, and can be merged. This allows the maximum use of available information in creating as accurate abstractions as possible at a given time.

PARSS uses progressive refinement. AS-UCT and ASAP-UCT, due to their complete recomputation of abstractions, allow both splits and merges. We know of no work that uses progressive abstractions, in part because there aren't many different abstraction procedures available.

Convergence Conditions & Units of Abstraction: A related question is "what does the final tree converge to?" Does it converge to a completely flat search space? Or does it converge to a reduced space? PARSS splits abstract states somewhat arbitrarily and, in the limit, converges to a perfectly flat search tree. On the other hand, AS- and ASAP-UCT use the definitions of node symmetries to converge to a reduced search space, which is guaranteed to be an accurate reduction of the MDP in the limit. Of these two, AS-UCT abstracts only the states, whereas ASAP-UCT can abstract state-action pairs as well, leading to more compression and runtime savings as shown before (Anand et al. 2015). PARSS operates only on states and not on state-action pairs.

Overall, we find that both PARSS and ASAP-UCT have some desirable and some undesirable properties. PARSS is an incremental and adaptive algorithm, but its abstractions are ad hoc and don't discover accurate domain symmetries. It also reduces to a flat space in the limit, and only aggregates states. ASAP-UCT, on the other hand, abstracts SAPs but is a batch-style algorithm and uniformly computes abstractions on the whole tree. Our proposal, OGA-UCT, combines the strengths of both algorithms: it computes SAP abstractions in an incremental and adaptive manner.

OGA-UCT

In this section, we describe our algorithm OGA-UCT. Our algorithm is best understood in terms of the construction of the original UCT tree. The UCT computation can be broadly divided in two key phases 1) Sampling of a trajectory 2) Random rollout from a newly discovered leaf node. In OGA-UCT, during the first phase, along with sampling of the trajectory, an abstraction for each node is also maintained *on-the-go*. Abstraction for any node is computed using the recursive updates similar to the ones used by ASAP-UCT (see the Background section). But the key difference is that instead of doing the batch computation uniformly for each node, we do it incrementally and in an adaptive manner. Each node has an associated recency count, which stores the number of times the node was visited after its abstraction was last updated. If the recency count reaches a pre-decided threshold K, we re-compute the abstraction for this node and set the recency count back to 0. In the second phase when a rollout is performed, we initialize the abstraction of the newly created leaf and set its recency count to 0. Any Q-value updates in the UCT tree are now done over the abstract nodes rather than the original nodes. This effectively means that we can utilize the information from a single rollout for all the nodes falling under the same abstraction. Since abstractions at a certain depth depend on the abstractions in the tree below, it may happen that when a node's abstraction changes, there is a change in the abstraction of its ancestor nodes. Therefore, any change in the abstraction of a node at depth d is propagated all the way up to the root of the tree, recomputing abstractions as necessary.

Our entire algorithm can be divided in four parts. 1) OGA-UCT procedure (Algorithm 1) which maintains the outer loop for sampling trajectories and is similar to the original UCT. 2) SampleTrajectory procedure (Algorithm 2) for sampling a single trajectory which is the key for OGA-UCT. 3) Procedures (Algorithm 3) for updating state and SAP abstractions and percolating the changes recursively to the ancestors once the recency count is reached for a node. 4) Procedures (Algorithm 4) for computing the current abstraction of a state and an SAP node. We next describe each one of them in detail.

Algorithm 1 OGA-UCT

```
1:  procedure OGA-UCT(S_0, N, K, Horizon)
2:      T ← EMPTYTREE()
3:      global K, Horizon, T
4:      Add state node (S_0, 0) to tree T
5:      INITIALIZESTATEABSTRACTION(S_0, 0)
6:      i ← 0
7:      while i < N do
8:          SAMPLETRAJECTORY(S_0, 0)
9:          i ← i + 1
10:     end while
11:     return SELECTBESTACTION(S_0, 0)
12: end procedure
```

OGA-UCT (Algorithm 1): The procedure OGA-UCT is very similar to the traditional UCT algorithm. We start from a root node $(S_0, 0)$ and then sample the required number of trajectories (N). Note that we need to initialize the abstraction of the root node (line 5). Horizon determines the depth until which the tree is expanded. K controls the frequency for computing abstraction; abstractions are recomputed when the recency count of a node becomes equal to K.

Sampling Trajectory (Algorithm 2): This is the main procedure of our algorithm. Lines 2-6 check the base condition for stopping a trajectory. Lines 7-11 add a newly discovered leaf node to the tree, create an abstract node for it(initialize its abstraction) and perform a rollout. If the procedure comes to line 12, we have not discovered a new leaf node yet. Line 12 selects an action based on the UCB rule. Here, Q-Values and Counts in UCB formula are obtained from Q-Values and Counts of corresponding abstract node. In lines 13-17, we add a newly discovered SAP node to the tree, create a new abstract node for it (initialize its abstraction) and set the recency count as 0. Lines 18-19 sample a new state node based on the chosen action and recursively call SAMPLETRAJECTORY. Lines 20-23 take care of maintaining the recency count and calling update abstractions if the count has reached the threshold K. Finally, lines 24-25 update counts and Q-Values for abstract node corresponding to (s, a, d). It is insightful to note that if we remove the lines for computing abstractions and maintaining the re-

cency count (lines 9,15-16,20-23), the procedure becomes identical to what standard UCT would do with lines 24, 25 updating Q-Value and Count of ground node.

Algorithm 2 Sample Trajectory in UCT

```
1: procedure VAL = SAMPLETRAJECTORY(s, d)
2:     if terminal(s) then
3:         return −reward(s)
4:     else if d == Horizon then
5:         return 0
6:     end if
7:     if (s, d) is not in tree T then
8:         Add state node (s, d) to tree T
9:         INITIALIZESTATEABSTRACTION(s, d)
10:        return GETROLLOUT(s, d)
11:    end if
12:    a ← SELECT-UCB-ACTION(s, d)
13:    if (s, a, d) is not in tree T then
14:        Add SAP node (s, a, d) to tree T
15:        INITIALIZE-SAP-ABSTRACTION(s, a, d)
16:        RecencyCount[s, a, d] ← 0
17:    end if
18:    s' ← SAMPLE(s, a)
19:    newVal ← SAMPLETRAJECTORY(s', d + 1)
20:    RecencyCount[s, a, d] + +
21:    if RecencyCount[s, a, d] == K then
22:        UPDATE-SAP-ABSTRACTION(s, a, d)
23:    end if
24:    INCREMENTCOUNT(μ^d_H(s, a))
25:    UPDATEQ(μ^d_H(s, a), newVal)
26:    return newVal
27: end procedure
```

Updating Abstractions (Algorithm 3): We update the abstractions for nodes whose recency count has reached K. There are two different procedures, one for updating SAP abstractions and other for updating state abstractions. The symbol $\mu^d_{\mathcal{E}}$ ($\mu^d_{\mathcal{H}}$) denotes the mapping from a state (SAP) node to its abstraction, as defined in the Background section. In the procedure for updating SAP abstractions, we first reset the recency count. Line 3 retrieves the current abstraction (v) of this node and line 4 computes the new abstraction (u). In lines 5 to 9, if the new abstraction is different from the old abstraction (i.e. $u \neq v$), the data of the old abstract node (v, d) and new abstract node (u, d) needs to be updated (since a ground node is leaving one and entering the other).The details about updating data(Q-Values and Counts) in line 7 is discussed in detail in sub-section on Maintaining Q-Values and Counts later. In line 8, update abstraction is then called on the parent node (s, d) to propagate up the influence of change in abstraction of node (s, a, d). The procedure for updating state abstraction (starting line 12) is similar. Since, a state node could have resulted from multiple (s', a') nodes, abstractions have to be updated for each one of them (lines 17 - 19). Note that we do not need to maintain any data for state nodes. The V-value of a state node is not explicitly required in UCT, and, the count of a state node is obtained by summing up the count of it's children SAP nodes.

Algorithm 3 Update Abstractions

```
1: procedure UPDATE-SAP-ABSTRACTION(s, a, d)
2:     RecencyCount[s, a, d] ← 0
3:     v ← μ^d_H(s, a)
4:     u ← COMPUTE-SAP-ABSTRACTION(s, a, d)
5:     if u ≠ v then                    ▷ if abstraction changed
6:         μ^d_H(s, a) ← u
7:         Update data of (v, d) and (u, d)
8:         UPDATE-STATE-ABSTRACTION(s, d)
9:     end if
10: end procedure
11: ——————————————————————————
12: procedure UPDATE-STATE-ABSTRACTION(s, d)
13:     y ← μ^d_E(s)
14:     x ← COMPUTE-STATE-ABSTRACTION(s, d)
15:     if x ≠ y then                    ▷ if abstraction changed
16:         μ^d_E(s) ← x
17:         for (s', a') ∈ Parents[s, d] do
18:             UPDATE-SAP-ABSTRACTION(s', a', d − 1)
19:         end for
20:     end if
21: end procedure
```

Computing Abstractions (Algorithm 4): There are two different procedures for computing abstractions: one for SAP nodes and one for state nodes. Let us look at the case of SAP nodes (Compute SAP Abstractions) first. Recall that $\mathcal{X}$ denotes the set of abstract state nodes at a given depth ($d+1$ in this case). Let $T_{\mathcal{X}}$ denote the vector of transition probabilities to abstract state nodes in depth $d+1$ from SAP node (s, a, d). Initially, these transition probabilities are set to 0 (line 1). In lines 3-5, $T_{\mathcal{X}}$ is populated by iterating over each node $(s', d+1)$ in the next level, finding its abstraction $\mu^{d+1}_{\mathcal{E}}$ and adding the transition probability $\mathcal{T}(s, a, s')$ to the corresponding element of vector $T_{\mathcal{X}}$. We also, maintain a hash map $M^d_{\mathcal{X}}$ which stores the mapping from the pairs of form $[T_{\mathcal{X}}, C(s, a)]$ to an abstract SAP node at depth d. If the key $[T_{\mathcal{X}}, C(s, a)]$ already exists in $M^d_{\mathcal{X}}$, the desired abstraction of (s, a, d) is the corresponding value (w in the line 7). Else we create a new SAP abstraction (u) containing (s, a, d) as the only node and add it to map $M^d_{\mathcal{X}}$ (lines 10,11).
Similarly for computing abstraction of state node (s) at depth (d), we maintain a set $\mathcal{J}_{\mathcal{U}}$ consisting of abstract SAP nodes of form (s, a, d) at depth d. Hash map $M^d_{\mathcal{U}}$ stores the mapping from set of form $\mathcal{J}_{\mathcal{U}}$ to the state abstractions at depth d. If $\mathcal{J}_{\mathcal{U}}$ already exists in hash-map, then we return the corresponding state abstraction else a new abstraction is created and returned.

Maintaining Q-Values and Counts: The ideal design principle would be to set the Q-value of an abstract SAP node as the weighted average of Q-values of the constituent nodes, and the associated count to be the sum of the constituent counts. Unfortunately, operationalizing this (at the time of abstraction change) requires significant bookkeeping. Hence, we maintain data(Q-values and Counts) only for abstract SAP nodes instead of individual nodes.

Let the abstraction of an SAP node (s, a, d) changes from v to u. Let the original counts for v and u be given by C_v and C_u respectively and new counts be given by C_v^{new} and C_u^{new} respectively. The new C_v^{new} and C_u^{new} can be computed in the following manner:

$$C_v^{new} = C_v - \frac{C_v}{|v|}, C_u^{new} = C_u + \frac{C_v}{|v|} \quad (1)$$

Intuitively, since we maintain count only for abstract SAP nodes, we take proportionate count from v and add it to u. Next, let Q_v and Q_u denote the original Q-values for v and u respectively. Then, the new Q-values, Q_v^{new} and Q_u^{new} can be computed by:

$$Q_v^{new} = Q_v; Q_u^{new} = \frac{C_u \cdot Q_u + \frac{C_v}{|v|} \cdot Q_v}{C_u + \frac{C_v}{|v|}} \quad (2)$$

Note that Q_v remains unchanged and Q_u is updated by taking a weighted average between Q_v and Q_u.

Algorithm 4 Compute-Abstraction

procedure COMPUTE-SAP-ABSTRACTION(s, a, d)
  $\forall x \in \mathcal{X} : \mathcal{T}_{\mathcal{X}}[x] = 0$
  for $(s', d+1)$ in Tree T **do**
    $\mathcal{T}_{\mathcal{X}}[\mu_{\mathcal{E}}^{d+1}(s')] += \mathcal{T}(s, a, s')$
  end for
  if $[\mathcal{T}_{\mathcal{X}}, C(s, a)]$ exists in $M_{\mathcal{X}}^d$ **then**
    $w = M_{\mathcal{X}}^d[\mathcal{T}_{\mathcal{X}}, C(s, a)]$
    return w
  end if
  $u \leftarrow$ CREATE-NEW-SAP-ABSTRACTION(d)
  Insert $[\mathcal{T}_{\mathcal{X}}, C(s, a)], u$ in $M_{\mathcal{X}}^d$
  return u
end procedure

procedure COMPUTE-STATE-ABSTRACTION(s, d)
  $\mathcal{J}_{\mathcal{U}} \leftarrow \{\}$
  for $a \in \mathcal{A}$ **do**
    $\mathcal{J}_{\mathcal{U}} \leftarrow \mathcal{J}_{\mathcal{U}} \cup \{\mu_{\mathcal{H}}^d(s, a)\}$
  end for
  if $\mathcal{J}_{\mathcal{U}}$ exists in $M_{\mathcal{U}}^d$ **then**
    $z = M_{\mathcal{U}}^d[\mathcal{J}_{\mathcal{U}}]$
    return z
  end if
  $x \leftarrow$ CREATE-NEW-STATE-ABSTRACTION(d)
  $M_{\mathcal{U}}^d[x] \leftarrow \mathcal{J}_{\mathcal{U}}$
  return x
end procedure

Pruned OGA-UCT: Many applications need to deal with domains with a very high stochastic branching factor. In such cases, OGA-UCT will spend a significant amount of time in computing SAP abstractions since large number of transitions need to be considered. In order to ameliorate this problem, while constructing transition tables, we don't consider nodes with very low transition probabilities. Assume that we need to compute abstraction of a SAP node (s, a, d). Let $\mathcal{T}_s^* = max_{s'}\mathcal{T}(s, a, s')$ where the maximisation is taken over the states s' such that $(s', d+1)$ is a node present in the UCT tree. Then, during abstraction computation, we only consider those nodes $(s', d+1)$ in the tree whose transition probability $\mathcal{T}(s, a, s') \geq \alpha * \mathcal{T}_s^*$. Here α is a constant s.t $0 \leq \alpha \leq 1$. We call the resulting algorithm Pruned OGA-UCT . Note that Pruned OGA-UCT defaults to OGA-UCT when $\alpha = 0$ As demonstrated by our experiments, Pruned OGA-UCT (for a suitably chosen value of α) is competitive with OGA-UCT while giving improved performance on domains with high branching factor.

Implementation Details: Whenever the abstraction of a state or SAP node changes, we might need to update the abstraction of its ancestors continuing all the way up to the root of the tree. Since UCT tree is a Directed Acyclic Graph, a single update of abstraction at a node may result in multiple such updates on an ancestor through different paths. In our implementation, we carefully avoid these multiple updates by performing them in a breadth first manner.

Characteristics of OGA-UCT

OGA-UCT has several desirable theoretical and algorithmic properties. We first prove that it converges to the optimal solution in the limit of infinite samples.

Theorem 1. *Given an MDP $M = (S, A, \mathcal{T}, C, H)$, the value function computed by OGA-UCT for the abstract node containing a state s at depth d converges to the value function computed by UCT for state s, as number of trajectories $N \to \infty$ i.e $\forall s \in S \ \ \forall d \leq Horizon$*

$$\lim_{N\to\infty} V_{OGA}^N(\mu_{\mathcal{X}}^d(s), d) = \lim_{N\to\infty} V_{UCT}^N(s, d)$$

Here V_{OGA}^N and V_{UCT}^N denote the value functions computed by OGA-UCT and UCT, respectively.

Proof Sketch: The proof will proceed in two parts.

Part 1: We say that abstractions computed by OGA-UCT are sound if two state (SAP) nodes that fall in the same abstraction under OGA-UCT, also fall in the same abstraction using the ASAP definition of abstractions in the ground finite-horizon MDP. Further, ASAP abstractions are guaranteed to have identical Q-values and V-values (Anand et al. 2015). Therefore, applying UCT on such a sound abstract tree will result in simulation of ground UCT which will converge to the optimal values in the limit.

Part 2: Let $N_{(s,d)}$ denote the number of trajectories passing through the state node (s, d). We say that (s, d) is visited sufficiently if $N_{s,d} \to \infty$ when $N \to \infty$. We define sufficient visits for SAP nodes in a similar manner. We will inductively prove that the abstractions in a sub-tree rooted at the state node (s, d) are sound if (s, d) is visited sufficiently. Let the D be the maximum depth in the tree. We will prove the claim by using backward induction from D going all the way to 0. Clearly, the claim is true for $d = D$ (leaves of the tree). Let us assume that it holds for state nodes at depth $d+1$. We will now prove it for depth d. Consider a state node (s, d). Since (s, d) is visited sufficiently, all its children SAP nodes must be in the tree (due to the exploration in UCT). Let (s, a, d) be one such child node. Let $u \in \mathcal{U}$ denote the abstract node corresponding to (s, a, d). Then, again due to

exploration in UCT, $\exists$ at least one node (s', a', d) with abstraction u which is visited sufficiently (all the nodes in an abstraction can not be starved). Since an SAP node samples its child nodes based on the transition probabilities, all its children $(s'', d+1)$ must be in the tree, must be visited sufficiently, and hence, should have sound abstractions using the inductive hypothesis. Combining the above two facts, we can say that (s', a', d) will also have a sound abstraction with any node in u. This implies that (s, a, d) (child of (s, a)) will have a sound abstraction. But since (s, a, d) was arbitrary, all the children of (s, d) must have sound abstractions. Combining this with the fact that all the children of (s, d) are already in the tree, (s, d) must also have sounds abstraction. Finally, since the root is visited sufficiently by the statement of the theorem, all the abstractions in the UCT tree must be sound in the limit. Hence, proved. □

We now place OGA-UCT into the context of our previous analysis of algorithmic design choices. OGA-UCT is *incremental* – it tightly integrates the abstraction computation routine with tree construction and makes only local changes in abstractions. Its focus on where to recompute abstractions is *adaptive* – it recomputes abstractions for frequently visited nodes much more often than others, thereby effectively utilizing the abstraction computation time on important parts of the search space. OGA-UCT can both *split* and *merge* existing abstractions, allowing itself to maintain as accurate a domain abstraction as possible given current knowledge. Last but not the least, it abstracts both states and state-action pairs, and in the limit converges to a reduced search space.

Experiments

We compare the performance of OGA-UCT with ASAP-UCT, the state-of-the-art UCT-based algorithm that employs domain abstractions. Previous work (Anand et al. 2015) showed that it obtains better performance than AS-UCT and variants. We also compare OGA-UCT with vanilla UCT to assess the overall value of abstractions in UCT. In addition, we also do a sensitivity analysis for different values of K in OGA-UCT. We illustrate our experiments on six popular MDP planning domains from literature and International Probabilistic Planning Competition (IPPC).

We implement OGA-UCT on the top of MDP Engine,[2] the UCT implementation from Bonet & Geffner (2012) in C++. ASAP-UCT[3] is also implemented over the same codebase. This makes the runtime comparisons between the three algorithms meaningful.

In spite of having access to PARSS source-code (Hostetler, Fern, and Dietterich 2015), we could not complete a meaningful comparison. PARSS is implemented in Java giving it a rather different execution profile. Its basis in sparse sampling and its unique style of abstractions makes PARSS very different in nature compared to UCT implementing AS, ASAP, or OGA. Because it starts with a tree pre-built up to the planning horizon, we do not expect PARSS performance to match up to UCT-based algorithms for large horizons – original PARSS experiments are on horizons of up to five. Their results are also on significantly modified versions of original benchmarks. We leave empirical comparison with PARSS to future work.

In our experiments, all algorithms were given equal time per decision, computed as total planning time divided by the execution horizon. UCT rollouts employed a random base policy. We set the exploration constant for UCB rule to be the absolute value of current Q-value of node, as per recommendations in (Bonet and Geffner 2012). The l value in ASAP-UCT, which determines the number of abstraction phases per decision, was set to 1 as per the recommendation in ASAP-UCT paper. We tried various K values and found OGA-UCT performance to vary slightly in a few domains, with no clear winner. We choose K to be 3 in all experiments for its marginally better overall performance.

All our experiments are performed on Intel Quad core i-7 system. For all the domains, we use a planning horizon of 50 and execution horizon of 100, i.e, a total of 100 decisions are taken per problem, and each decision is taken with a maximum lookahead of 50.

Domain Descriptions

We briefly describe the MDP domains used in our expts.

Race Track: This traditional MDP (Barto, Bradtke, and Singh 1995; Bonet and Geffner 2012) consists of a race track with acceleration and deceleration in either direction as the available actions. We test on six different race-tracks as implemented in Bonet's MDP Engine.

SysAdmin: We use IPPC 2011 (Sanner and Yoon 2011) version of this traditional domain (Guestrin et al. 2003). The agent is a network administrator, and can reboot machines. Machines can probabilistically crash based on the number of alive machines that are neighbors. We test on six problems – ring, hub, and line topologies with 10 and 15 machines each.

Navigation: In this IPPC 2011 domain the goal is to navigate a grid, but with a probability, the agent teleports back to the starting cell. The probabilities in different cells vary, making it possible for the agent to find longer paths, which actually have a higher probability of reaching the goal. We test on five IPPC 2011 instances of varying sizes.

Sailing Wind: In this popular grid world domain (Kocsis and Szepesvári 2006; Bonet and Geffner 2012) each grid element has additional information about wind direction. An agent can move in seven directions (except opposite to wind direction), but pays variable cost based on the wind direction. We use 5 different instances having grid sizes $\{10, 15, 20, 25, 30\}$ in our tests.

Academic Advising: This domain from IPPC 2014 requires an agent to pass various courses that have pre-requisite relations (Guerin et al. 2012). Good grades in pre-requisites makes it more likely to pass a course. We test on 4 different IPPC instances for this domain.

Game of Life: In this IPPC 2011 gridworld domain, cells can die due to overpopulation or underpopulation depending upon number of alive neighbors. An agent can make a cell alive every time step. The domain has a very high branching factor. We test on two instances for grid sizes 3×3, 4×4, and 5×5 with uniform noise probabilities in each cell.

[2] Available at https://code.google.com/p/mdp-engine/

[3] Downloaded from https://github.com/dair-iitd/asap-uct

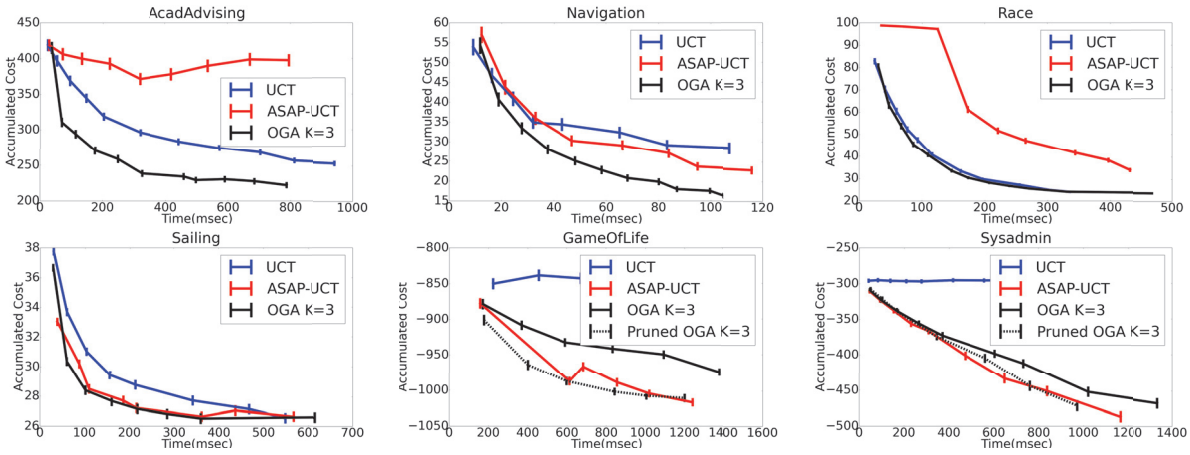

Figure 1: OGA-UCT performs better or at par with ASAP-UCT and UCT for most of the domains

Domains	UCT	ASAP-UCT	OGA (K=3)	Pruned ASAP	Pruned OGA (K=3)
Acadadvising	0.48 ± 0.07	0.23 ± 0.16	$\mathbf{0.89 \pm 0.03}$	0.23 ± 0.16	$\mathbf{0.88 \pm 0.03}$
Navigation	0.41 ± 0.16	0.45 ± 0.13	$\mathbf{0.57 \pm 0.13}$	0.45 ± 0.12	$\mathbf{0.54 \pm 0.08}$
RaceTrack	$\mathbf{0.84 \pm 0.13}$	0.38 ± 0.09	$\mathbf{0.84 \pm 0.13}$	0.46 ± 0.11	$\mathbf{0.82 \pm 0.14}$
Sailing Wind	0.61 ± 0.03	$\mathbf{0.80 \pm 0.05}$	$\mathbf{0.82 \pm 0.05}$	$\mathbf{0.82 \pm 0.04}$	$\mathbf{0.82 \pm 0.03}$
GameOfLife	0.14 ± 0.17	$\mathbf{0.81 \pm 0.06}$	0.64 ± 0.11	0.44 ± 0.21	$\mathbf{0.82 \pm 0.04}$
Sysadmin	0.02 ± 0.01	$\mathbf{0.66 \pm 0.03}$	0.54 ± 0.01	0.45 ± 0.05	0.57 ± 0.08

Table 2: Aggregate performance across different problems and planning times per domain normalized between 0 and 1. Pruned OGA-UCT is best or on par with the best on almost all domains, including those with high branching factors.

Observations

We compare OGA-UCT with ASAP-UCT and unabstracted UCT on these problems with different total planning times and draw cost vs. time curves. Representative runs on each domain are illustrated in Figure 1. Each curve is an average of 1,000 reruns and also draws 95% confidence interval bars.

In addition to the representative curves we also show aggregate performance of an algorithm on each domain across different planning times (Table 2). We choose six equispaced planning times for each problem and run the three algorithms along with all variants (pruned versions and different K values). Of all of these points, we give the least cost a score of 1, and the worst cost a score of zero. We normalize each cost value to a number between 0 and 1. This normalization is related to the metric used by IPPCs, with small differences. For example, IPPCs normalize only across algorithms and not across planning times.

We observe that OGA-UCT performs the best or on par with the best on four out of the six domains. In AcadAdvising and Navigation, OGA's performance is substantially better than both algorithms. It matches performance with UCT on RaceTrack and with ASAP-UCT on Sailing Wind. In Game of Life (GOL) and Sysadmin, OGA-UCT performs worse than ASAP-UCT. Both of these domains have exponential branching factors, which severely slow down abstraction computation routines. Since OGA recomputes abstractions more often than ASAP, it suffers significantly.

The pruned version of OGA-UCT helps with exactly this (we set probability threshold to 0.1). Pruned OGA-UCT makes abstraction computations approximate and only higher probability transitions are taken into account while computing abstractions. This improves performance on both domains, with pruned OGA-UCT becoming at par with ASAP-UCT on GOL. The performance in other domains remains mostly similar. For fair comparison, we also attempt pruning with ASAP-UCT and find that it hurts substantially in GOL and Sysadmin. Since ASAP-UCT computes abstractions only a handful of times per decision step, computing them accurately is likely more important for it than the computational savings due to approximation.

Overall, we find that performances of ASAP-UCT and UCT can depend heavily on the domain, but OGA-UCT admits least variance and is robustly good across several domains.[4]

Sensitivity Analysis across different K-Values: We ran all our experiments for $K = 1, 3, 5, 7$. Table 3 shows the aggregate score for different Ks. While there is no single K, which is best across all domains, most of the performances are significantly close to each other. We choose $K = 3$, which gives an overall balanced performance across all domains.

[4]Our comparative results between ASAP-UCT and UCT are different from those in Anand *et al.* (2015). This is because we identified a bug in Bonet's MDP Engine, and fixing that bug changed performances of all algorithms slightly. This bug was verified by Bonet.

Dom.	OGA (K=1)	OGA (K=3)	OGA (K=5)	OGA (K=7)
Acad.	0.86 ± 0.02	0.89 ± 0.03	0.91 ± 0.03	0.87 ± 0.08
Nav.	0.73 ± 0.07	0.57 ± 0.13	0.50 ± 0.11	0.45 ± 0.13
Race.	0.79 ± 0.17	0.84 ± 0.13	0.85 ± 0.12	0.85 ± 0.13
Sail.	0.76 ± 0.03	0.82 ± 0.05	0.84 ± 0.03	0.83 ± 0.03
GOL	0.62 ± 0.09	0.64 ± 0.11	0.63 ± 0.11	0.61 ± 0.11
Sys.	0.42 ± 0.02	0.54 ± 0.01	0.59 ± 0.02	0.61 ± 0.02

Table 3: Comparison of OGA-UCT different K-values for all domains. Performances remain similar, and there is no clear winner.

Conclusions

We present OGA-UCT, an algorithm to compute domain abstractions on the go within the UCT framework. It makes several desirable design choices such as it computes abstractions of state-action pairs, using an incremental and adaptive computation of abstractions, with a tight coupling between abstraction computation and tree construction. This allows OGA-UCT to efficiently recover from inaccurate abstractions as more information gets available. In the limit of infinite samples, OGA-UCT obtains a sound reduction of the original search tree and converges to the optimal solution.

Our experiments demonstrate that OGA-UCT is robust across domains. It compares favorably to the best of the algorithms in many domains. However, it can suffer when the branching factor is very high because that directly impacts the abstraction computation routine. An extension of OGA-UCT that prunes various low-probability transitions allows it to scale to such domains. Overall, Pruned OGA-UCT obtains best performance in almost all domains obtaining up to 28% quality gains.

We also contribute our analysis of algorithmic design choices applicable to MCTS with abstractions. We hope that this analysis will be useful in understanding existing algorithms and also for algorithm development in the future.

Acknowledgements

We thank Alan Fern for his valuable comments, which resulted in our exploring this research direction. We thank Blai Bonet for his help with the MDP engine. We also thank the anonymous reviewers for their in-depth reviews, which helped improve the paper significantly. Ankit Anand is supported by TCS Research Scholars Program. Mausam is supported by Google and Bloomberg research awards.

References

Anand, A.; Grover, A.; Mausam; and Singla, P. 2015. ASAP-UCT: Abstraction of State-Action Pairs in UCT. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, 1509–1515.

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

Bellman, R. 1957. A Markovian Decision Process. *Indiana University Mathematics Journal*.

Bonet, B., and Geffner, H. 2012. Action Selection for MDPs: Anytime AO* Versus UCT. In *AAAI*.

Givan, R.; Dean, T.; and Greig, M. 2003. Equivalence notions and model minimization in Markov decision processes. *Artificial Intelligence* 147(12):163 – 223.

Guerin, J. T.; Hanna, J. P.; Ferland, L.; Mattei, N.; and Goldsmith, J. 2012. The academic advising planning domain. *WS-IPC 2012* 1.

Guestrin, C.; Koller, D.; Parr, R.; and Venkataraman, S. 2003. Efficient solution algorithms for factored MDPs. *J. Artif. Intell. Res.(JAIR)* 19:399–468.

Hostetler, J.; Fern, A.; and Dietterich, T. 2014. State Aggregation in Monte Carlo Tree Search. In *AAAI*.

Hostetler, J.; Fern, A.; and Dietterich, T. 2015. Progressive abstraction refinement for sparse sampling. In *Conference on Uncertainty in Artificial Intelligence (UAI)*.

Howard, R. A. 1960. Dynamic programming and markov processes.

Jiang, N.; Kulesza, A.; and Singh, S. 2015. Abstraction selection in model-based reinforcement learning. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, 179–188.

Jiang, N.; Singh, S.; and Lewis, R. 2014. Improving UCT Planning via Approximate Homomorphisms. In *AAMAS*.

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

Keller, T., and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *ICAPS*.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Machine Learning: ECML*. Springer.

Li, L.; Walsh, T. J.; and Littman, M. L. 2006. Towards a Unified Theory of State Abstraction for MDPs. In *ISAIM*.

Mausam, and Kolobov, A. 2012. *Planning with Markov Decision Processes: An AI Perspective*. Morgan & Claypool Publishers.

Ravindran, B. 2004. *An Algebraic Approach to Abstraction in Reinforcement Learning*. Ph.D. Dissertation, University of Massachusetts Amherst.

Sanner, S., and Yoon, S. 2011. International Probabilistic Planning Competition (IPPC) 2011. In *ICAPS*.

Srinivasan, S.; Talvitie, E.; and Bowling, M. 2015. Improving Exploration in UCT Using Local Manifolds. In *AAAI Conference on Artificial Intelligence*.