Bootstrapping Simulation-Based Algorithms with a Suboptimal Policy

Truong-Huy Dinh Nguyen[*]
Colleges of Arts, Media and Design
Northeastern University
Boston, MA 02115, USA
tru.nguyen@neu.edu

Tomi Silander
Xerox Research Centre Europe
38240 Meylan, France
tsilande@xrce.xerox.com

Wee-Sun Lee, Tze-Yun Leong
Department of Computer Science
National University of Singapore
Singapore 117417, Singapore
{leews,leongty}@comp.nus.edu.sg

Abstract

Finding optimal policies for Markov Decision Processes with large state spaces is in general intractable. Nonetheless, simulation-based algorithms inspired by Sparse Sampling (SS) such as Upper Confidence Bound applied in Trees (UCT) and Forward Search Sparse Sampling (FSSS) have been shown to perform reasonably well in both theory and practice, despite the high computational demand. To improve the efficiency of these algorithms, we adopt a simple enhancement technique with a heuristic policy to speed up the selection of optimal actions. The general method, called *Aux*, augments the look-ahead tree with *auxiliary* arms that are evaluated by the heuristic policy. In this paper, we provide theoretical justification for the method and demonstrate its effectiveness in two experimental benchmarks that showcase the faster convergence to a near optimal policy for both SS and FSSS. Moreover, to further speed up the convergence of these algorithms at the early stage, we present a novel mechanism to combine them with UCT so that the resulting hybrid algorithm is superior to both of its components.

Introduction

Markov Decision Processes (MDPs) provide a common framework to model complex planning problems. Many recent approaches have been proposed to solve large MDPs, where the state spaces may be exponential in size with respect to the numerous factors or relations that characterize real-life applications. In discounted-reward problems, i.e., situations where immediate rewards/costs are valued exponentially more than rewards/costs further in future, one viable approach to solve large MDPs is by constructing a randomly sampled look-ahead tree that covers only a small fraction of the state space to estimate the Q-values achievable by the applicable actions starting in a given state. This random sampling approach to estimation requires a simulator, or a *generative model*, of the environment. The approach results in near-optimal policies that are independent of the state space size, but at the cost of an exponential dependence on the *horizon time*, determined by the reward discount factor and the desired degree of proximity to the optimal policy (Kearns, Mansour, and Ng 2002). This dependency is proven to be close to the best dependency achievable by any planning algorithm that is given only a simulator. Exponential run-time, therefore, is still needed for deriving reasonable policies for MDPs with discount factors close to 1. In real life applications, these algorithms are often enhanced with heuristics in order to approach optimal values faster (Gelly and Silver 2007).

Nguyen et al. (Nguyen, Lee, and Leong 2012) recently proposed to use a heuristic policy to bootstrap simulation-based algorithms for faster convergence to near-optimal policies. This general method, called *Aux* for auxiliary policy estimation, augments the look-ahead tree with additional arms that are evaluated by a heuristic policy π. The value of the augmented arm is estimated from the average reward of multiple π-guided rollouts from a given state. When applied to the widely used Monte Carlo Tree Search algorithm namely Upper Confidence Bound applied in Trees (UCT) (Kocsis and Szepesvári 2006), the proposed *UCT-Aux* algorithm was shown to yield performance improvement when the heuristic deployed in π satisfies certain conditions (Nguyen, Lee, and Leong 2012). UCT, however, is inherently optimistic in selecting simulated actions and focuses the simulations in highly rewarding branches exponentially more often than others. Therefore, *UCT-Aux* and the other UCT variants tend to converge to the optimal policy very slowly; the rate can be super-exponentially slow in some cases.

In this paper, we investigate the auxiliary estimation approach in other more conservative simulation-based algorithms for large MDPs. In particular, we examine the Aux-method coupled with Sparse Sampling (SS) (Kearns, Mansour, and Ng 2002), the backbone of many modern simulation-based algorithms, and Forward Search Sparse Sampling (FSSS) (Walsh, Goschin, and Littman 2010), one of the state-of-the-art sparse sampling algorithms. While there have been various efforts in enhancing the performance of UCT (Gelly and Silver 2007; Finnsson and Björnsson 2008; Chaslot et al. 2010; Nguyen, Lee, and Leong 2012; Keller and Helmert 2013), to the best of our knowledge, our work is among the first to investigate possible enhancement for the algorithms addressed. In subsequent sections, we provide theoretical evidence that SS-Aux and FSSS-Aux

[*] A majority part of this work was completed when the author was with the National University of Singapore.

yield performance improvements in run-time efficiency over SS and FSSS. We then demonstrate empirically that both SS-Aux and FSSS-Aux outperform their respective vanilla versions by converging faster to near-optimal policies in two benchmark domains. In the experiments, UCT-Aux is faster than SS-Aux and FSSS-Aux in finding a near optimal policy, but FSSS-Aux eventually beats UCT-Aux in performance. This motivates a hybrid approach that dynamically switches between UCT-Aux and FSSS-Aux algorithms yielding performance superior to both of them.

Background

Markov Decision Processes (MDP)

An MDP characterizes a planning problem with the tuple (S, A, T, R, γ), in which

- S is the state set, and A is the set of available actions.
- $T(s, a) = P(s_{t+1} \mid s_t = s, a_t = a)$ is a probability distribution over next states when action a is taken in state s at time t; we assume that this distribution is time-independent.
- $R(s, a)$ is the immediate reward received after executing action a in state s. We assume that $R(s, a)$ is deterministic and bounded, i.e., there is $R_{max} \in \mathbb{R}$ such that $|R(s, a)| \leq R_{max}$ for all $s \in S$ and $a \in A$.
- $\gamma \in [0, 1)$ is the discount factor to downplay temporally remote rewards.

A *policy* π is a possibly stochastic function that returns an action $\pi(s) \in A$ for every state $s \in S$. In infinite-horizon, discounted MDPs, the value function V and *Q-function* of a policy π measure the expected long-term utility of following π:

$$\begin{aligned} V^\pi(s_0) &= \mathbb{E}_T\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t))\right], and \\ Q^\pi(s_0, a) &= R(s_0, a) + \gamma \underset{s' \sim T(s_0, a)}{\mathbb{E}} [V^\pi(s')]. \end{aligned}$$

The objective of solving MDPs is to obtain a policy π^* with the maximum expected values in all the states, i.e., $\forall s, \pi V^*(s) \geq V^\pi(s)$. One common approach is to compute the optimal Q-function Q^* and construct the corresponding policy $\pi^*(s) = \text{argmax}_a Q^*(s, a)$.

Regret of a policy

Given a policy π of an MDP $M = (S, A, T, R, \gamma)$, we can define its performance guarantee as follows:

Definition 1 *A policy π is called **ϵ-optimal** if its **regret**, $\max_{s \in S} V^*(s) - V^\pi(s)$, is at most ϵ.*

Note that a policy's regret never exceeds $2V_{max}$ with $V_{max} = \frac{R_{max}}{1-\gamma}$.

Sampling π to estimate $V^*(s)$

Most simulation-based algorithms aim to approximate the optimal value $V^*(s) = \max_a Q(s, a)$ for the current state s. The Aux enhancement's key idea is to opportunistically choose between the value estimation of the core algorithm and that of the heuristic policy at each tree node. The latter, when empirically yielding higher value estimates, is the main source of estimation improvement.

We will now study the error in approximating $V^*(s)$ if we simply simulate a sub-optimal policy π for B times for L steps, and use the average accumulative reward as an estimate. In this so-called *π-estimate*, the total estimation error stems from three sources: (1) the regret of π, (2) the length L of rollouts, and (3) the limited number B of samples. The following lemma[1] bounds the error in estimating $V^*(s)$ using π-guided rollouts.

Lemma 1 *Let π be an ϵ_0-optimal policy, $X_{\pi,L,i}$ be the accumulative reward of π-guided rollout i from state s and $\bar{V}^\pi_{B,L}(s) = \frac{1}{B}\sum_{i=1}^{B} X_{\pi,L,i}$. For all $p > 0$, with probability at least $1 - p$,*

$$|V^*(s) - \bar{V}^\pi_{B,L}(s)| \leq \epsilon_0 + \gamma^L V_{max} + O\left(V_{max}\sqrt{\frac{\ln(1/p)}{B}}\right).$$

Proof Sketch. By using triangle inequality we get

$$\begin{aligned} |V^*(s) - \tilde{V}^\pi_{B,L}(s)| \leq\ & |V^*(s) - V^\pi(s)| \\ & + |V^\pi(s) - V^\pi_L(s)| \\ & + |V^\pi_L(s) - \tilde{V}^\pi_{B,L}(s)| \end{aligned}$$

with $V^\pi_L(s) = \mathbb{E}(X_{\pi,L,i})$ for all i. By definition, the first term is bounded by ϵ_0. The second term is the expected estimation error for V^π due to simulating π only for L steps; this is bounded by $\gamma^L V_{max}$. Finally, using Hoeffding inequality with B i.i.d. and bounded values $X_{\pi,L,i}$, we bound the last term with probability at least $1 - p$. □

By increasing the values of B and L, we can get as close as desired to the ϵ_0-vicinity of a state's value with high probability, at the cost of longer running time that scales with BL.

Bootstrapping Sparse Sampling

The Sparse Sampling (SS) (Kearns, Mansour, and Ng 2002) algorithm estimates the best action to take in a state s_0 based on a set of random simulations of policy "rollouts" or action execution sequences. The approach is motivated by the observation that for discounted reward functions the knowledge about rewards and transition probabilities in a vicinity of the current state suffices for decision making; events far in future are irrelevant due to the discount factor γ. Consequently, a sufficiently extensive simulation of the near future would adequately represent the transition probabilities and the expected cumulative rewards.

The Sparse Sampling Algorithm

The SS algorithm (Algorithm 1 called with $h = H$ and aux=False) estimates the state-action values $Q^*(s_0, a)$ of the optimal policy by constructing a look-ahead tree rooted at

[1] All the lemmas and propositions in this paper are our contributions; the detailed proofs are however omitted due to space constraints.

s_0 at level H (see Figure 1a). Starting from s_0, it simulates each of the $k = |A|$ possible actions C times in the depth-first postorder down to level 0 where all values $V_0^{SS}(s)$ are set to some default value, e.g., 0. After the subtrees at level $h-1$ have been sampled, the node values are backed up to level h by $Q_h^{SS}(s,a) = R(s,a) + \frac{\gamma}{C}\sum_{c=1}^{C} V_{h-1}^{SS}(s'_c)$, where s'_c denotes a state at level $h-1$ that was obtained when simulating action a in state s the c^{th} time. The V-values of states at each level are based on the same level Q-values by $V_h^{SS}(s) = \operatorname{argmax}_a Q_h^{SS}(s,a)$.

```
Input : state $s$, height $h$, boolean flag $aux$
if $h = 0$ or Terminal (s) then
    $V_0^{SS}(s) \leftarrow 0$
end
else if $\neg Visited(s,h)$ then
    $Visited(s,h) \leftarrow$ true
    foreach $a \in A$ do
        $Q_h^{SS}(s,a) \leftarrow 0$
        for $C$ times do
            $s' \sim T(s,a)$
            $Q_h^{SS}(s,a) \leftarrow Q_h^{SS}(s,a) + \frac{1}{C}R(s',a)$
            $+\frac{\gamma}{C}$ SS (s',h-1,aux)
        end
    end
    $V_h^{SS}(s) \leftarrow \max_a Q_h^{SS}(s,a)$
    if $aux$ and $\bar{V}^{\pi}_{L,B}(s) > V_h^{SS}(s)$ then
        $V_h^{SS}(s) \leftarrow \bar{V}^{\pi}_{L,B}(s)$
    end
end
return $V_h^{SS}(s)$
```

Algorithm 1: SS (s, h, aux)

As detailed above, the value-estimates of leaf nodes in a SS look-ahead tree were set to zero. In general, given a leaf value estimation $V_0^{SS}(s)$ that is ϵ_0-optimal, i.e., $\forall s, |V^*(s) - V_0^{SS}(s)| \le \epsilon_0$, the value estimation error at the root node s_0 is composed of two factors: one due to the limited sampling parametrized by H and C, and one due to the leaf error ϵ_0. The following lemma shows that as H and C increases, with C kept at some magnitude larger than H, and ϵ_0 is bounded, the algorithm's regret $\epsilon_{SS} = \max_s |V^*(s) - V^{SS}(s)|$ goes to 0.

Lemma 2 *Consider SS with leaf nodes' values being ϵ_0-optimal, i.e. $\forall s, |V^*(s) - V_0^{SS}(s)| \le \epsilon_0$. We have*

$$\epsilon_{SS} \le T_{SS}(H,C) + \frac{\gamma^H \epsilon_0}{1-\gamma} \tag{1}$$

with $T_{SS}(H,C) = \frac{2\gamma^H V_{max}}{1-\gamma} + \frac{V_{max}}{(1-\gamma)^2}\sqrt{\frac{H}{C}\ln\frac{kC}{\gamma}}$.

In the bound, the first error term $T_{SS}(H,C)$ is due to limited sampling, while the second partly depends on the quality of leaf estimation ϵ_0. If high values of H, C are affordable, the leaf estimation plays little role in the algorithm's quality.

Specifically, Kearns et al. (Kearns, Mansour, and Ng 2002) showed that by setting the leaf values to 0, i.e., leaf regret $\epsilon_0 = V_{max}$, and H and C dependent on $\epsilon > 0$ such that $C = O\left(H^2/\gamma^{2H}\right)$, the resultant estimation of the algorithm is ϵ-optimal.

SS-Aux: Sparse Sampling with π-guided Auxiliary Arms

Given a deterministic heuristic policy π, the proposed algorithm, called *SS-Aux* (Algorithm 1 called with aux=True), adds one auxiliary arm into each internal state node of the look-ahead tree[2]. The auxiliary arm stemming from s is labeled by the action $a = \pi(s)$ (Figure 1b) and it does not expand to subtrees as do its siblings, but its value is computed as the average of accumulated rewards returned by π-guided simulation traces (Section "Sampling π to estimate $V^*(s)$"). The auxiliary arm is treated equally to its ordinary siblings, i.e., the value of the parent node is computed by taking the maximum of all child nodes' estimated values. Note that the number of auxiliary arms increases exponentially with the height of the tree if these arms are added to every internal node. Therefore, in order to manage the incurred extra cost, we can stop adding auxiliary arms below a certain height $\hat{H}$ with $\hat{H} < H$.

Unlike SS which completely relies on the value estimation of expanded subtrees, whenever the heuristic π appears to yield higher estimated value than any of its siblings, the parent node's value is estimated by this heuristic value instead. Eventually, this improved estimation propagates and contributes to the root's value. Intuitively, if the heuristic is good at states that are near the root on the optimal path, the improvement is instantiated early in the process. Consequently, SS-Aux can be run with shorter height H, which saves computing resources.

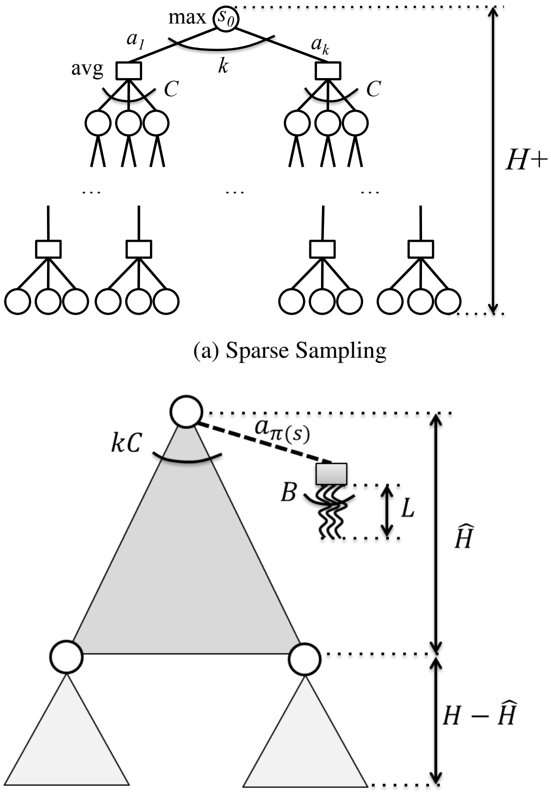
While the running time of SS is $(kC)^H$, SS-Aux's is $(kC)^H + LB(kC)^{H-\hat{H}} = (kC)^H(1 + LB/(kC)^{\hat{H}})$. Depending on the resources allocated, $\hat{H}$ can be adjusted so that $LB/(kC)^{\hat{H}}$ is negligible, in which case the running time of SS-Aux is asymptotically equal to that of SS.

Bootstrapping Forward Search Sparse Sampling

SS-Aux suffers from the impracticality of SS due to unfocused sampling which allocates a fixed amount of computation for every tree branch and node. As a result, SS requires a long running time before delivering any meaningful approximation. The problem is more pronounced in domains that are highly stochastic and where discount factor is close to 1.

There have been many attempts such as Heuristic Sampling (Péret and Garcia 2004), Adaptive Sampling (Chang et al. 2005) and Upper Confidence Bound applied in Trees (UCT) (Kocsis and Szepesvári 2006) to improve the practicality of the forward sampling approach. All the forward

[2] The method can be generalized to stochastic policies by adding κ auxiliary arms to each state node s, with κ being the number of actions a such that $P(\pi(s) = a) > 0$.



(a) Sparse Sampling

(b) SS-Aux

Figure 1: The look-ahead trees of SS and SS-Aux; each state node (circles) estimates V_h^{SS} while its state-action children (rectangles) estimate Q_h^{SS} for each height $h \in [H - \hat{H}, H]$.

sampling methods use a similar tree-like structure as SS, so there is potential to enhance them using the Aux method. The latest in this series is the Forward Search Sparse Sampling (FSSS) (Walsh, Goschin, and Littman 2010) that tries to smartly allocate the computation to promising branches of the look-ahead tree in order to more quickly approximate the optimal policy in large domains. In this section, we investigate the application of the Aux method to enhance FSSS.

Forward Search Sparse Sampling (FSSS)

FSSS is the latest successor of SS that is able to combine the performance guarantee of SS with selective sampling (Walsh, Goschin, and Littman 2010). Instead of directly forming point estimates of the Q- and V-values in a look-ahead tree, it maintains interval estimates that allow it then to guide the sampling to the promising and/or unexplored branches of the look-ahead tree based on the widths and upper bounds of the intervals.

Similar to SS, FSSS also constructs a tree of height H and branching factor k. However, instead of constructing the whole tree in one go, FSSS traverses the tree from root down using simulated episodes of length H called *trials*. At each state node s of the trial, the algorithm *expands* the state by simulating all the k actions C times. Instead of then recursing to all the next states like SS, the FSSS picks the action a^* with a highest upper-bound for Q(s,a) and *selects* the next state s^* with the widest interval for V(s') among the states s' that were obtained by simulating a^* at s. The selected state s^* is then expanded until the leaf node is reached, after which the interval bounds of V- and Q-values are *updated* recursively on the path from leaf to root. When there is an action at the root with the Q lower bound greater than all other arms' upper bounds, i.e., the action is surely more highly rewarding than its siblings, the algorithm terminates.

As such, FSSS may also skip subtree expansions at branches where no further exploration would be helpful. When no pruning occurs, FSSS requires at most $(kC)^H$ trials. However, its running time may be up to $O(H(kC)^H)$.

Proposition 1 *The running time of FSSS is bounded by* $O(H(kC)^H)$.

Proof Sketch. At each non-leaf state node, *expansion* samples the simulator kC times, thus incurring a cost of $O((kC)^H)$ in total. In one FSSS trial, *selection* and *update* take $O(H)$, so since there are at most $(kC)^H$ trials, altogether these two actions cost $O(H(kC)^H)$. The total running time is therefore dominated by $O(H(kC)^H)$. □

This means that in the worst case FSSS may require longer running time than SS.

FSSS-Aux: FSSS with π-guided auxiliary arms

Similar to SS-Aux, we add one auxiliary arm at every internal state node of the tree up to a certain depth. At each of these auxiliary arms, the lower and upper bounds are computed using B π-guided rollouts of length L. The procedure is detailed in Algorithm 2 with the *aux* flag set to true and $EstimateQ(s, a, \pi)$ being the average bounds returned by π-guided simulations. Similar to the behavior of SS-Aux, it is straightforward to show that in the worst case, when auxiliary arms do not help to prune any branch, FSSS-Aux yields the same estimation as that of FSSS. In the good case when the heuristic policy is near-optimal, we expect many state nodes have their lower-upper bound gap closed faster. In fact, the following proposition states that, after terminating, FSSS-Aux is as good as SS-Aux.

Proposition 2 *On termination, the action chosen by FSSS-Aux is the same as that chosen by SS-Aux.*

Proof sketch. Note that if there is no pruning possible, FSSS-Aux expands the same tree as SS-Aux. Otherwise, upon termination, there is no point for further tightening the bounds because one arm surely has higher value than the rest, as its lower bound exceeds the upper bounds of other arms. This means that the selected arm is guaranteed to be the same as the one selected by SS-Aux, which expands the full tree. □

Note that however at premature termination, there is no guarantee on the performance of FSSS as compared to that of SS (or FSSS-Aux to SS-Aux for that matter) due to their dissimilarity in exploring the state space. Specifically, FSSS uses upper-lower bound gap to guide the search, while SS distributes sampling equally among all child nodes to collect rewards' statistics. As such, they may discover the optimal values at different time points.

```
Input : state $s$, height $h$
if $h = 0$ or Terminal(s) then
    $L^h(s), U^h(s) \leftarrow 0, 0$
    return
end
if $\neg Visited(s, h)$ then
    $Visited(s, h) \leftarrow$ true
    foreach $a \in A$ do
        $L^h(s, a), U^h(s, a) \leftarrow V_{min}, V_{max}$
        $S_h(s, a) \leftarrow \{\}$
        for $C$ times do
            $s' \sim T(s, a)$
            $C_h(s, a, s') \leftarrow C_h(s, a, s') + 1$
            $S_h(s, a) \leftarrow S_h(s, a) \cup \{s'\}$
            $L^{h-1}(s'), U^{h-1}(s') \leftarrow V_{min}, V_{max}$
        end
    end
    $\hat{a} \leftarrow \pi(s)$        /* Auxiliary arm */
    $L^h(s, \hat{a}), U^h(s, \hat{a}) \leftarrow$ EstimateQ$(s, \hat{a}, \pi)$
end
$a^* \leftarrow \text{argmax}_a U^h(s, a)$
$s^* \leftarrow \text{argmax}_{s'} C_h(s, a^*, s')(U^{h-1}(s') - L^{h-1}(s'))$
FSSS-Aux-Rollout$(s^*, h - 1)$
$L^h(s, a^*) \leftarrow R(s, a^*) + \frac{\gamma}{C} \sum_{s'} C_h(s, a^*, s') L^{h-1}(s')$
$U^h(s, a^*) \leftarrow R(s, a^*) + \frac{\gamma}{C} \sum_{s'} C_h(s, a^*, s') U^{h-1}(s')$
$L^h(s) \leftarrow \max_a L^h(s, a)$
$U^h(s) \leftarrow \max_a U^h(s, a)$
return
```

Algorithm 2: FSSS-Aux-Rollout (s, h)

Complexity-wise, the number of trials required by FSSS-Aux has the same bound as that of FSSS.

Proposition 3 *The total number of trials of FSSS-Aux before termination is bounded by the number of leaves in the tree.*

Proof sketch. The proof is identical to that of Proposition 3 by Walsh et al. (Walsh, Goschin, and Littman 2010), by noting that every trial of FSSS-Aux has to end at a node with zero bound gap. If such a node is not a leaf, it would not be selected in the first place, because it cannot be the node with widest bound gap. As such, every trial must end at a leaf node, i.e., the number of FSSS-Aux rollouts is bounded by the number of leaves in the tree. □

Similar to the situation with SS-Aux, since the additional time required by FSSS-Aux is that of evaluating auxiliary arms, with suitable values of $\hat{H}$, this incurred cost is negligible, rendering the running time of FSSS-Aux being asymptotically comparable to that of vanilla FSSS, i.e. $O(H(kC)^H)$. Nevertheless, in the good case when the heuristic policy is near-optimal, we expect the algorithm to terminate faster and produce a better approximation than FSSS, as the heuristic helps discover the optimal arm earlier.

Numerical Experiments

In this section, we will assess the performance of the Aux method when used with UCT, SS and FSSS in two benchmark domains, Obstructed Sailing and Sheep Farmer. As compared to other popular bootstrapping methods with UCT (Gelly and Silver 2007), UCT-Aux is known to be very competitive if the coupled heuristic is "extreme", but underperforms otherwise due to the inherent optimism of UCT in distributing rollouts (Nguyen, Lee, and Leong 2012). As such, we would also like to observe how SS-Aux and FSSS-Aux fare with UCT-Aux in cases when UCT-Aux stumbles.

Note that for SS algorithms, instead of setting a fixed value for the tree height H, we implement them using Iterative Deepening with respect to H. These online versions of SS are allocated a certain amount of time for each planning step, starting out with an initial small value of H. Upon termination, if there is still time for planning, H is increased and the algorithm is restarted. However, the sampling width factor C is kept fixed. Even though the theoretical bound is established with C magnitudes larger than H, empirically a value for C roughly reflecting the stochasticity of the domain suffices.

Obstructed Sailing

In the Obstructed Sailing domain (Nguyen, Lee, and Leong 2012), the agent tries to move a boat, in the most cost-effective manner, around obstacles from a starting point to a destination in changing wind conditions that affect the cost of execution. Note that the domain's stochasticity comes from the fact that the wind direction frequently changes according to a probabilistic transition matrix. In our experiments, after each time step, the wind has roughly equal probability to remain unchanged, switch to its left or its right (Kocsis and Szepesvári 2006). The maps used are of size 20 by 20 with starting position at (5, 5) and goal at (15, 15), and the obstacles are placed with probability 0.4 at each grid square. The algorithms are pitted in 1000 different random maps for which the benchmarked optimal policy is obtained by Value Iteration (Bellman 1957).

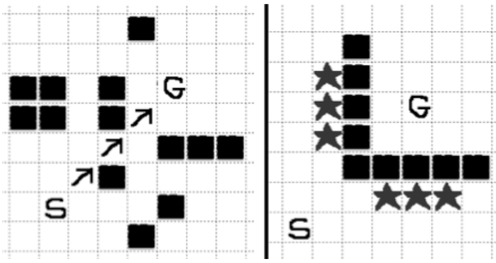


Figure 2: Sails-To-Goal heuristic produces near-optimal estimates/policies in good cases (left) but misleads the search control in others (right).

In bootstrapping the algorithms, we use two heuristics that show different behavior in terms of performance: Sails-To-Goal and Stochastic Optimal.

Sails-To-Goal (STG). This heuristic selects a valid action closest to the direction towards goal position regardless

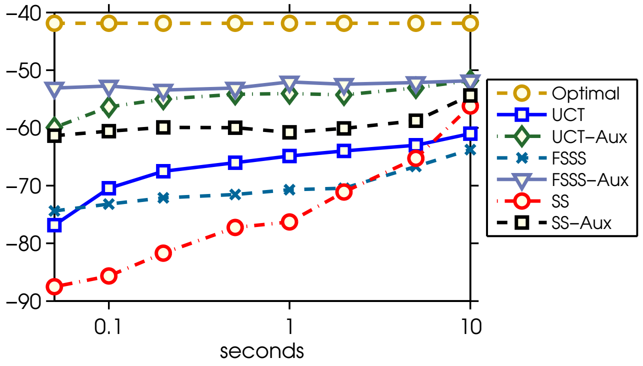


Figure 3: Aux algorithms in Obstructed Sailing when coupled with SailsToGoal. y-axis is the negative cost average (the standard errors of the means are smaller than the markers' size), while x-axis shows, in log scale, the allocated time per planning step in seconds.

of the cost (Nguyen, Lee, and Leong 2012). It is therefore particularly good for maps with few obstacles, producing near-optimal plans for many states, but counterproductive with highly obstructed maps, since it often gets stuck in dead ends and corners that surround the goal position (Figure 2). As such, this heuristic shows behaviors of "*make it or break it*" (*extreme*), i.e., in most states s, the action value estimate $Q_\pi(s,a)$ is either near-optimal or as low as that of a random policy $Q_{rand}(s,a)$. In Obstructed Sailing, STG yields an average accumulated cost of 161, far from the optimal cost of 41.88.

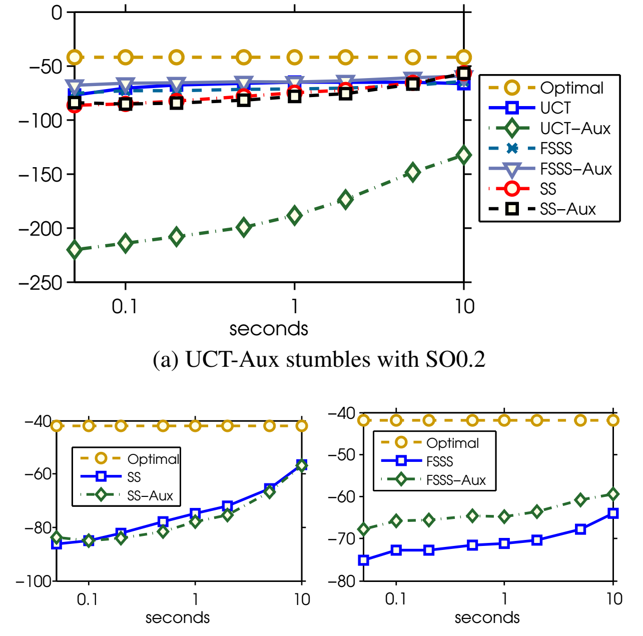
Stochastic Optimal with $p = 0.2$ (SO0.2). This heuristic is an example for heuristics that are milder in nature than Sails-To-Goal. It issues optimal actions with probability $p = 0.2$ and random actions for the rest. This policy is also suboptimal but almost always yields better estimation than random movement; it is not as "*extreme*" as Sails-To-Goal. However, the highly randomized nature of the policy causes it to stumble badly too; the average accumulated cost of SO0.2 is 256.66.

Results. Figure 3 and 4 depicts the average negative costs of UCT, SS and FSSS and their Aux-bootstrapped versions when coupled with two π-heuristics, STG and SO0.2 respectively. As explained above, STG is known to be "extreme" and preferable by UCT-Aux, while SO0.2 is not, causing UCT-Aux to stumble badly. Indeed, UCT-Aux outperforms vanilla UCT with STG (Figure 3) but is completely outmatched with SO0.2. In contrast, SS-Aux and especially FSSS-Aux manage the heuristics much better, yielding improved performance over their respective host algorithms.

Sheep Farmer

Sampling-based algorithms are designed to tackle the problem of intractability in exact methods such as Value Iteration and Policy Iteration in large domains. In this section, we examine how SS, FSSS and UCT perform in a collaborative game called a Sheep Farmer that has over 10^{11} states.

In this game domain, the planner controls a team of a farmer and his dog to herd a sheep into a pen and kill two wolves in a maze-like environment (Nguyen, Lee, and Leong 2012); the sheep and wolves are hereafter referred to as non-player characters (NPCs). All NPCs run away from the farmer and dog when close to them, otherwise the wolves chase the sheep and the sheep runs away from wolves; all movements succeed with probability 0.9. Both farmer and dog have 5 moves (no-move, N, S, E and W) but the farmer has an additional action to inflict damage on a nearby wolf, hence a total of $6 \times 5 = 30$ joint actions. The farm-team is given 5 points for successfully killing wolves and 10 points for herding sheep into its pen. In the event that the sheep is killed, the game ends with penalty -10. We used discount factor $\gamma = 0.99$. This domain serves as an example of two-player collaborative games.



(a) UCT-Aux stumbles with SO0.2

(b) SS-Aux

(c) FSSS-Aux

Figure 4: Aux algorithms in Obstructed Sailing when coupled with StochOpt0.2.

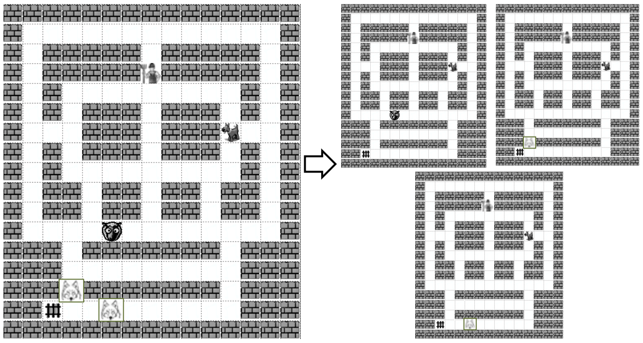


Figure 5: Task decomposition in Sheep Farmer; each task involves dealing with either one wolf or one sheep.

We ran SS, FSSS and UCT side by side with their respective Aux-bootstrapped versions[3] in the map shown in Figure 5. The optimal policy in this domain was not computed due to the prohibitively large state space of size $104^5 * 3^2 \approx 10^{11}$ (each wolf has two health points). For fairer comparison SS agents were implemented using the Iterative Deepening approach with respect to height H of the look-ahead tree. Each algorithm was allocated a fixed amount of time per planning step; due to time constraints, we did not run simulations with more than 50 seconds per planning step. In a real time game domain like this, the region of interest is typically from less than 1 to 5 seconds. The experiment was run 200 times, each with a different randomized starting position of the characters.

Constructing a heuristic policy The Aux method requires a heuristic policy for the auxiliary arms; one way to construct a heuristic policy in this domain is via *task decomposition*. In particular, the game can be seen as having three subtasks: killing one of the two wolves or herding the sheep, as shown in Figure 5. Each of these subtasks consists of only the farmer, the dog and one of the NPCs, hence the subtasks are small enough to be solved offline using Value Iteration. This yields Q-values for the subtasks, and a heuristic Q-value for the whole game can now be obtained by taking the average of the subtasks' Q-values. Each state s of the game can be projected to a state s_i of the i^{th} subtask by ignoring the NPCs not in the subtask i. We can then form a goal averaging (GA) heuristic via $Q_{GA}(s,a) = \frac{1}{m}\sum_{i=1}^{m} Q_i(s_i,a)$ with m being the number of subtasks. The corresponding heuristic policy is then defined as $\pi_{GA}(s) = \text{argmax}_a\, Q_{GA}(s,a)$.

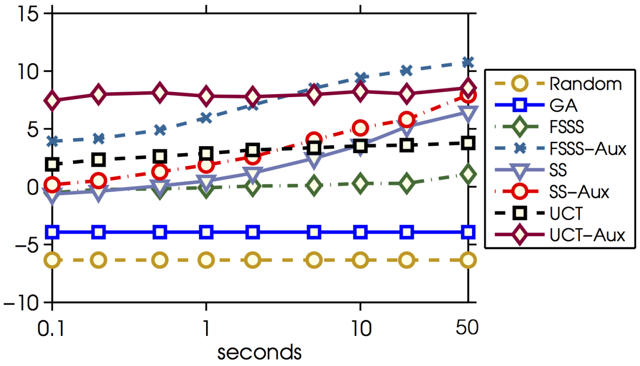


Figure 6: SS, FSSS and UCT variants in Sheep Farmer with GoalAveraging-heuristic. The charted lines denote the average accumulated reward of the algorithms in 200 experiments as function of planning time (displayed in log scale).

Experiment results Figure 6 shows how the average accumulated rewards of SS, FSSS and UCT variants, the Random-action policy and our GoalAveraging-heuristic vary with respect to the allocated planning time.

Even when coupled with a relatively poor (reward -4) heuristic π_{GA}, both SS and FSSS improve over their original vanilla versions. Moreover, while vanilla FSSS exhibits much slower convergence to the optimal policy than SS, FSSS-Aux convincingly dominates all other methods. Between two state-of-the-art algorithms UCT and FSSS, UCT-Aux shows improvements much earlier than FSSS-Aux, but is eventually outperformed by the latter.

This result is the evidence of how differently UCT and FSSS choose to explore the state space. While UCT's optimism, which distributes an exponentially higher amount of simulations on currently best-rewarding branches, slows down its discovery of optimal actions if they lie deep down the search tree, FSSS takes a more conservative approach which focuses on branches that are most uncertain. This approach slows down the process of discovering highly rewarding branches of vanilla FSSS as compared to that of UCT, but plays a crucial role in enabling continuing improvement of FSSS-Aux, as it keeps exploring and improving even when a local minimum is discovered by the heuristic, as long as there are still uncertain regions in the search tree.

Combining UCT and FSSS

In order to get the good performance of UCT initially and FSSS eventually, we introduce a hybrid algorithm that is constructed on top of UCT and FSSS with the aim to combine the best parts of both algorithms. Details are furnished in Algorithm 3.

```
Input  : state $s$, boolean flag $aux$
1  $s_{UCT} \leftarrow$ UCTNode $(s, aux)$
2  $\mathcal{T} \leftarrow \emptyset$
3  $s_{FSSS} \leftarrow$ FSSSNode $(s, aux)$
4  while still has time and $L^H(s_{FSSS}) < U^H(s_{FSSS})$ do
5      $b \sim Ber(H_N(\mathcal{T}))$   /* Sample the choice
       of algorithm */
6      if $b = 1$ then
7          UCT-Aux-Rollout $(s_{UCT}, H)$
8          $\mathcal{T} \leftarrow \mathcal{T} \cup$ LastRollout-RootArm $(s_{UCT})$
9      end
10     else
11         FSSS-Aux-Rollout $(s_{FSSS}, H)$
12     end
13 end
14 $a_{FSSS}, V_{FSSS} \leftarrow$ argmax$_a$, max$_a L^H(s_{FSSS}, a)$
15 $a_{UCT}, V_{UCT} \leftarrow$ argmax$_a$, max$_a Q_{UCT}(s_{UCT}, a)$
16 if $V_{FSSS} > V_{UCT}$ then
17     return $a_{FSSS}$
18 end
19 return $a_{UCT}$
```

Algorithm 3: UCT-FSSS(s, aux)

The idea is to keep track of the distribution p of selected simulated arms at the root node of UCT tree and sample UCT less frequently (Line 5) when the choice of arm at the root node has stabilized. We formalize the uncertainty about

[3] The implementation of *UCT-Aux* is the one used by Nguyen et al. (Nguyen, Lee, and Leong 2012).

actions at the UCT tree root $\mathcal{T}$ using the normalized entropy

$$H_N(\mathcal{T}) = \frac{-\sum_{a\in A} p(a)\log(a)}{\log(|A|)} \in [0,1]. \quad (2)$$

The distribution p is initialized to the uniform distribution by sampling all actions at the root node $\mathcal{T}$ once. After that the distribution is always updated by normalizing the numbers of times the different actions at the root node are sampled. As shown in the algorithm, the more stable UCT's choice of root arms is, the more likely FSSS is picked (Line 5); the distribution shown is Bernoulli distribution with parameter $H_N(\mathcal{T})$. At termination, the algorithm returns the arm that yields highest approximated value among UCT and FSSS arms; we use lower bounds as the approximated values for FSSS arms.

The advantages of this mechanism are twofold

1. It never stops sampling UCT and FSSS trees; as such, the convergence properties of both algorithms are retained. That means our hybrid algorithm will converge to the optimal solution, given enough computational resources.
2. It flexibly focuses less or more simulations on UCT depending on whether UCT has started to converge. If $H_N(\mathcal{T})$ becomes large again, i.e., UCT escapes a local optimum, the algorithm will sample UCT more often.

Note that the algorithm is not limited to UCT and FSSS only; it is applicable to any two sampling-based algorithms. The adaptivity in algorithm selection depends on the convergence of the algorithms.

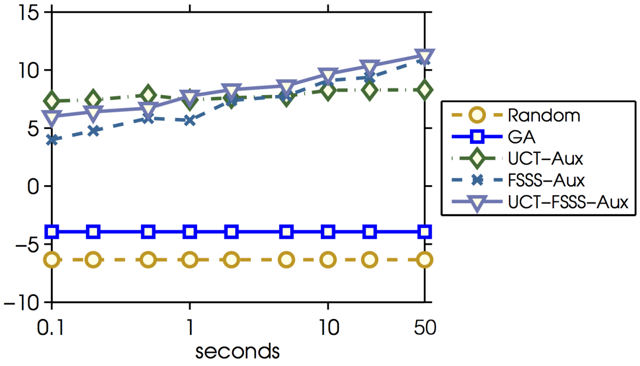


Figure 7: Average accumulated rewards of FSSS-Aux, UCT-Aux and the hybrid algorithm of UCT-Aux and FSSS-AUx in Sheep Farmer.

Figure 7 demonstrates the performance of the hybrid algorithm as compared with its components, i.e., UCT-Aux and FSSS-Aux, in the Sheep Farmer setup as described in Section . As we can see, the hybrid approach is able to mitigate the slow head start of FSSS-Aux by closing the gap with UCT-Aux more quickly, while eventually matching FSSS-Aux's performance and outperforming UCT-Aux.

Notably, in Obstructed Sailing, when coupled with SO0.2, the hybrid shows to compensate the blunder of UCT-Aux by leaning more towards FSSS-Aux, as UCT-Aux starts to converge on highly sub-optimal auxiliary arms guided by the heuristic. The average accumulated costs are depicted in Figure 8, with optimal cost being 41.88.

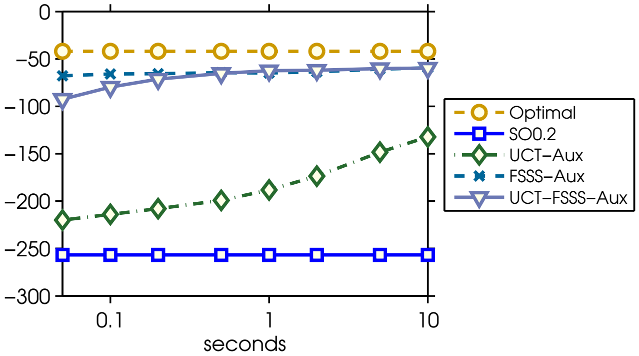


Figure 8: Average accumulated negative cost of UCT-Aux, FSSS-Aux and Hybrid (UCT-Aux and FSSS-Aux) when coupled with SO0.2 in Obstructed Sailing; the hybrid is marginally affected by the underperformance of UCT-Aux.

Conclusion

In this work, we have examined the Aux technique of bootstrapping simulation-based algorithms SS and FSSS with a suboptimal heuristic policy. When coupled with suitable heuristics, the technique was shown to yield significant improvement over the host algorithms in approximating the optimal action policy, while incurring only negligible overhead. Among the bootstrapped algorithms, we observed that FSSS-Aux, inheriting a more conservative approach in exploration from FSSS than that of UCT, is the preferred algorithm for solving large state-space MDPs when there are abundant computing resources available. On the other hand, when the allocated resources are scarce, UCT and its variants are able to discover a reasonably near-optimal solution more quickly. Alternatively, one can adopt a hybrid approach and combine these two sampling-based algorithms to obtain an algorithm that inherits the good bits of them both: quick discovery of local optima and steady convergence to the global optima. Notably, the hybrid variant of UCT-Aux and FSSS-Aux can also mitigate the under-performance of UCT-Aux when coupled with unfavorable heuristics. In the future we would like to investigate the application of Aux-bootstrapped algorithms in other contexts than MDPs such as adversarial settings.

Acknowledgments

This work was partially supported by Academic Research Grants: T1 251RES1005, MOE2010-T2-2-071, and T1 251RES1211 from the Ministry of Education, Singapore. The authors would also like to thank Northeastern University for partially supporting the work of the first author.

References

Bellman, R. 1957. *Dynamic Programming*. Princeton, NJ: Princeton University Press.

Chang, H. S.; Fu, M. C.; Hu, J.; and Marcus, S. I. 2005. An adaptive sampling algorithm for solving Markov Decision Processes. *Operations Research* 53(1):126–139.

Chaslot, G.; Fiter, C.; Hoock, J. B.; Rimmel, A.; and Teytaud, O. 2010. Adding expert knowledge and exploration in Monte-Carlo Tree Search. *Advances in Computer Games* 1–13.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to General Game Playing. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI '08)*, 259–264. AAAI Press.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning (ICML '07)*, 273–280. New York, NY, USA: ACM.

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

Keller, T., and Helmert, M. 2013. Trial-based Heuristic Tree Search for Finite Horizon MDPs. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS '13)*, 135–143.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In *Proceedings of the 17th European conference on Machine Learning (ECML '06)*, ECML'06, 282–293. Berlin, Heidelberg: Springer-Verlag.

Nguyen, T.-H. D.; Lee, W.-S.; and Leong, T.-Y. 2012. Bootstrapping Monte Carlo Tree Search with an Imperfect Heuristic. In *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD'12)*, 164–179. Berlin, Heidelberg: Springer-Verlag.

Péret, L., and Garcia, F. 2004. On-line search for solving Markov Decision Processes via heuristic sampling. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004)*, 530–534.

Walsh, T. J.; Goschin, S.; and Littman, M. L. 2010. Integrating sample-based planning and model-based reinforcement learning. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*.