Online Symbolic Gradient-Based Optimization for Factored Action MDPs

Hao Cui and Roni Khardon
Department of Computer Science, Tufts University, Medford, Massachusetts, USA
Hao.Cui@tufts.edu, roni@cs.tufts.edu

Abstract
This paper investigates online stochastic planning for problems with large factored state and action spaces. We introduce a novel algorithm that builds a symbolic representation capturing an approximation of the action-value Q-function in terms of action variables, and then performs gradient based search to select an action for the current state. The algorithm can be seen as a symbolic extension of Monte-Carlo search, induced by independence assumptions on state and action variables, and augmented with gradients to speed up the search. This avoids the space explosion typically faced by symbolic methods, and the dearth of samples faced by Monte-Carlo methods when the action space is large. An experimental evaluation on benchmark problems shows that the algorithm is competitive with state of the art across problem sizes and that it provides significant improvements for large factored action spaces.

1 Introduction
Recently there is increased interest in stochastic planning in domains with large combinatorially structured action spaces. Such structure arises, for example, when multiple units can operate in parallel to achieve a joint objective. Such problems have been modeled using factored Markov decision processes (MDP) where state and action descriptions are composed of the values of sets of variables. Factored MDPs have been investigated with a variety of approaches which are reviewed in the next section including symbolic dynamic programming, probabilistic inference, Monte Carlo Search, linear function approximation and more. However, apart from function approximation with manually tuned representations, scalability to large action spaces is still a significant challenge. Notably, methods that rely on exact representation of value functions are space bound due to the inherent complexity of representing the corresponding functions. On the other hand symbolic search based methods are not able to explore the space sufficiently when given a strict time limit for action selection.

The main contribution of this paper is a novel algorithm that offers such scalability. The algorithm is based on the idea of Monte Carlo search with a strong simplifying assumption on independence of state and action variables. Such assumptions are common in variational inference [Wainwright and Jordan, 2008], and have been used for estimation in state space models [Murphy and Weiss, 2001] even for planning [Lang and Toussaint, 2010]. Our recent work [Cui et al., 2015] made the observation that, under this assumption, sampling of trajectories can be done in aggregate form by rewriting the transition model as algebraic expressions and then computing marginals on state variables by direct calculation instead of explicit sampling. While this efficiently samples trajectories for a given root action it requires action enumeration and does not scale for large action spaces.

In this paper we make two additional observations. First, the algebraic computation can be done symbolically. We show how the sampled trajectories and the corresponding marginals can be captured using an explicit graph representing the computation, where the state marginals and the reward are functions of action variables. Second, given this graph, we can calculate gradients and perform gradient ascent over the action space, effectively optimizing the action for the current state. Several additional algorithmic ideas make for a robust search algorithm that dynamically adjusts to the problem and balances the benefits of Monte Carlo search and gradients. An extensive experimental evaluation demonstrates that the algorithm performs significantly better than previous work and that all its components contribute to that success.

2 Background and Related Work
This section gives a brief review of factored MDPs and related algorithms. A MDP [Puterman, 1994] is specified by \( \{S, A, T, R, \gamma\} \), where \( S \) is a finite state space, \( A \) is a finite action space, \( T(s, a, s’) = p(s’|s, a) \) defines the transition probabilities, \( R(s, a) \) is the immediate reward of taking action \( a \) in state \( s \), and \( \gamma \) is the discount factor. A policy \( \pi : S \rightarrow A \) is a mapping from states to actions, indicating which action to choose at each state. Given a policy \( \pi \), the value function \( V^\pi(s) \) is the expected discounted total reward \( E^\pi \sum \gamma^i R(s_i, \pi(s_i)) \mid s_0 = s \), where \( s_i \) is the \( i \)th state visited by following \( \pi \) (and \( s_0 = s \)). The action-value function \( Q^\pi : S \times A \rightarrow R \) is the expected discounted total reward when taking action \( a \) at state \( s \) and following \( \pi \) thereafter.

MDPs with large state and action spaces are typically represented in factored form [Boutilier et al., 1995a]. The state
space $\mathcal{S}$ is specified by a set of binary variables $\{x_1, \ldots, x_l\}$ so that $|\mathcal{S}| = 2^l$. Similarly, $\mathcal{A}$ is specified by a set of binary variables $\{y_1, \ldots, y_m\}$ so that $|\mathcal{A}| = 2^m$. Often, when domains are described using a higher level language, action constraints limit the choice of legal actions so that $|\mathcal{A}| < 2^m$.

Basic MDP optimization algorithms require enumeration over states and actions making them impractical for large problems. Symbolic versions of these algorithms have been developed for factored state and action spaces [Hoey et al., 1999; Boutilier et al., 1995b; Raghavan et al., 2012; 2013]. However, scalability is still problematic due to the description size of value functions and policies. Parametric methods represent the value function using some compact representation (e.g., as a linear function over a fixed feature space) and thus avoid the space problem. Such approaches have been very successful in reinforcement learning [Tesauro, 1992] and in policy gradient algorithms [Williams, 1992].

Online planning avoids the space complexity by focusing on choosing a high-quality action for the current state and repeatedly applying this procedure to states visited. The Rollout Algorithm [Tesauro and Galperin, 1996], which performs one step lookahead from the current state, is the simplest such procedure. In particular, given a state $s$ and baseline policy $\pi$, the rollout algorithm calculates an estimate of $Q^\pi(s,a)$ for each action $a$ and then picks the $a$ maximizing this value. To estimate $Q^\pi(s,a)$ simply use the MDP model to sample $s'$ from $p(s'|s,a)$ and then apply $\pi$ for $h$ steps where $h$ is the desired horizon. Repeating this and averaging we get an estimate for $Q^\pi(s,a)$. More sophisticated online planning methods include real time dynamic programming [Barto et al., 1995] and Monte Carlo Tree Search (MCTS) [Brown et al., 2012] and the Gourmand [Kolobov et al., 2012] and PROST [Keller and Helmert, 2013] systems based on these have won top places in recent planning competitions.

The performance of sample based Rollout procedures degrades significantly when the action space is large because one has to sample multiple trajectories for each action in order to get reasonable estimates. With combinatorial action spaces there are very few samples per action and the estimates have high variance and are not reliable. The ARollout algorithm [Cui et al., 2015] is a heuristic that attempts to address this issue. Since our algorithm uses some of the ideas from ARollout we describe it in more detail in the next section.

### 2.1 The ARollout Algorithm

We illustrate the ARollout algorithm using the following example. In this example, as in our implementation, we use the RDDL language [Sanner, 2010] to specify the domain. The problem has three state variables $s(1), s(2)$ and $s(3)$, and three action variables $a(1), a(2), a(3)$ respectively. In addition we have two intermediate variables $cond1$ and $cond2$ which are not part of the state. The dynamics is given by the following expressions where primed variants of variables represent the value of the variable after performing the action.

\[
\begin{align*}
cond1 &= \text{Bernoulli}(0.7) \\
cond2 &= \text{Bernoulli}(0.5) \\
s'(1) &= \text{if} (\text{cond1}) \text{ then } \neg a(3) \text{ else } \text{false} \\
s'(2) &= \text{if} (s(1)) \text{ then } a(2) \text{ else } \text{false} \\
s'(3) &= \text{if} (\text{cond2}) \text{ then } s(2) \text{ else } \text{false} \\
\text{reward} &= s(1) + s(2) + s(3)
\end{align*}
\]

ARollout translates the RDDL code into algebraic expressions using standard transformations from a logical to a numerical representation. In our example this yields:

\[
\begin{align*}
s'(1) &= (1-a(3)) \times 0.7 \\
s'(2) &= s(1) \times a(2) \\
s'(3) &= s(2)/2 \\
r &= s(1) + s(2) + s(3)
\end{align*}
\]

These expressions are used for efficient approximation of marginal distributions over state variables, and the distribution is used to estimate the $Q$ function. Like Rollout, to estimate $Q^\pi(s,a)$ ARollout first samples $s'$ from $p(s'|s,a)$. However, unlike Rollout, instead of simulating individual trajectories, ARollout approximates the simulation of state distributions through their marginals over individual variables.

To illustrate, let the sampled $s'$ be $s' = \{s(1)=1, s(2)=1, s(3)=1\}$. ARollout first translates $s'$ into a representation of marginals over individual variables, referred to as aggregate states and denoted as. In this case, \(as = \{s(1)=1, s(2)=1, s(3)=1\}\). ARollout then calculates marginals over action variables which are induced by using the rollout policy $\pi$ in the current aggregate state $as$. This can be estimated by first sampling a concrete state $cs$ from $as$, then calculating the action $ca = \pi(cs)$ and averaging over action variable values. For our example, assume that this yields $a_{00} = \{a(1)=0.3, a(2)=0.4, a(3)=0.3\}$. The reward $r_0$ from the current state and action is estimated by directly plugging in the values of each variable into the expression for $r$. So we get $r_0 = 1 + 1 + 1 = 3$. Similarly, $as_1$ is obtained by substituting marginals into the expressions for state variables. In this case, $as_1 = \{s'(1)=(1-0.3) \times 0.7=0.49, s'(2)=1 \times 0.4=0.4, s'(3)=0.3\}$. The algorithm continues in this manner, computing aggregate actions, aggregate states and estimating the reward to the required depth. This provides one aggregate trajectory. This simulation is correct under extreme assumptions [Cui et al., 2015] and it can be seen as a heuristic estimate when the conditions do not hold.

ARollout repeats this to obtain multiple aggregate trajectories per action and averages the reward to get $Q^\pi(s,a)$. This process is repeated for every action $a$, and the maximizing $a$ is picked to be used in the current state. ARollout’s estimates are more stable than Rollout’s estimates but the need to enumerate actions limits applicability to large action space.

### 3 Gradient Based Optimization

The main contribution of this paper is to provide an algorithm that scales successfully for factored action spaces. The algorithm is based on two important observations. The first is that the calculation illustrated in the previous section can be performed symbolically to yield a compact expression capturing the approximation of the $Q$ function as a function of action variables. The second is that, once such a representation exists, one can use gradient based optimization to search for the best action in the current state. Several additional ideas are used to make the search more robust and to diversify its exploration. We denote this algorithm as SOGBOFA (symbolic online gradient based optimization for factored actions).

The input to SOGBOFA is the same as in ARollout. As explained below, we assume that constraints over legal actions,
if they exist, are given as sum constraints (sum of all or some variables is bounded, e.g., concurrency constraints). Unlike ARollout, SOGBOFA maintains an aggregate action $aa$ for the current state. This represents a distribution over actions to be chosen in the current state. The algorithm is as follows:

**SOGBOFA(S)**

```plaintext
1  Qf ← BuildQf(S, timeAllowed)
2  As = { }
3  while time remaining
4    do A ← RandomRestart()
5      while time remaining and not converged
6        do D ← CalculateGradient(Qf)
7        A ← MakeUpdates(D)
8        A ← Projection(A)
9        As.add(SampleConcreteAct(A))
10   action ← Best(As)
```

**Overview of the Algorithm:** In line 1, we build an expression graph that represents the approximation of the $Q$ function. This step also explicitly optimizes a tradeoff between simulation depth and run time to ensure that enough updates can be made. Line 4 samples an initial action for the gradient search. Lines 6 to 8 calculate the gradient and make an update on the aggregate action. Line 9 makes the search more robust by finding a concrete action induced by the current aggregate action and evaluating it explicitly. Line 10 picks the action with the maximum estimate. Line 5 checks our stopping criterion which allows us to balance gradient and random exploration. In the following we describe these steps in more details.

**Building a symbolic representation of the $Q$ function:** We build a DAG representing the $Q$ function by expanding from the current state one level at a time. The depth of this process, which is equivalent to the search depth in standard algorithms, is determined dynamically as explained below. The construction is analogous to the construction of expression trees in a compiler. Leaf nodes represent either constants or state or action variables. Internal nodes represent algebraic operations over the children. Our implementation uses $+, -, *, /$ as operations but it can be easily extended to include other operations. The implementation directly follows the algebraic expressions compiled from the RDDL description (as in ARollout) to build explicit expression DAGs. The only exception is the flattening of multiple levels of $+$ or $*$; for example $(+(+(a, b)), c)$ is flattened to $+(a, b, c)$.

We start from representing each initial state variable as a leaf node whose value is 0 or 1. We also represent each action variable at the initial state as a leaf node. In this case the node represents the marginal probability of that variable, serving as a parameter to be optimized. For each step of expansion, we create a copy of the algebraic expressions in DAG form. To simulate the rollout, action variables other than at the root are given a numerical value capturing the marginal probability over the action variable induced by $\pi$. The marginals can be calculated dynamically exactly as in ARollout. In the experiments in this paper we report on rollout of the random policy where marginals can be calculated explicitly even with sum constraints.\(^{1}\) If the MDP domain requires discounting we add this to the expression tree. The sum of discounted rewards is the estimate of the $Q$ value. Figure 1 illustrates the construction for our example for horizon 1. For deeper horizons this construction is repeated sequentially.

We note that action variables at levels 2 and above (nodes with value 1/3 in the figure) can also be treated as parameters, exactly as the variables of the first level. This leads to an algorithm related to conformant planning. We discuss this idea further in the concluding section of the paper.

**Dynamic control over simulation depth:** In principle we should build the DAG to the horizon depth. However, if the DAG is large then evaluation of $Q^* (s, a)$ and gradient computation are expensive so that the number of actions explored in the search might be too small. This tradeoff must be controlled by any domain independent online search algorithm.

Our algorithm limits depth when the expected number of gradient updates is below a pre-fixed threshold. This can be done dynamically. In particular we estimate if the time left is sufficient for building one more level of the DAG and making enough updates. The time cost of adding one level to the DAG, $t_a$, is independent of the level and can be estimated by the average of the previous levels. The time cost of an update at the current level, $t_I$, is estimated by simulated updates on the current DAG (we use $\min\{0.3 * NumberOfLegalAction, 5\}$ updates). We estimate $t_{I+1}$, by $t_I + \Delta t$ where $\Delta t$ is the average increase in cost over previous levels. Finally, we expand one more level if $t_a + k * t_{I+1}$ is smaller than the remaining time, guaranteeing at least $k$ expected updates (where $k = 200$ in our experiments).

**Random Restarts:** A random restart generates a concrete (binary) legal action in the state.

**Calculating the gradient:** The main observation is that once the graph is built, representing $Q$ as a function of action variables, we can calculate gradients using the method of automatic differentiation [Griewank and Walther, 2008]. This is a

\[^{1}\text{For example, if the constraints choose at most } k \text{ bits out of } n \text{ to be true we can use} \ p = \sum_{j=1}^{k} (j/n) * \binom{n}{j}/(\sum_{j=0}^{k} \binom{n}{j}).\]
standard method that has also been recently used in machine learning. Our implementation uses the reverse accumulation method since it is more efficient having linear complexity in the size of the DAG for calculation of all gradients. Intuitively, the idea can be seen to generalize the backpropagation algorithm for learning in neural networks to an arbitrary computational circuit. Since the ideas are well known we refer the reader to [Griewank and Walther, 2008] for details.

**Maintaining action constraints:** Gradient updates allow the values of marginal probabilities to violate the \([0, 1]\) constraints as well as constraints on legal actions. To handle this issue we use a standard algorithm, projected gradient ascent [Shalev-Shwartz, 2012], where one first takes a gradient step and then projects the values back to the legal region.

Projection onto combinatorial constraints can be computationally expensive. Our implementation limits allowed constraints to sum constraints over action variables. This is typical, for example, in high level representations that use 1-of-k representation for actions where at most one bit among a group of \(k\) should be set to 1. It is important to enforce these constraints during search. If this is not done then, if all actions are somewhat useful, we will end up with the uninformative choice of setting all variables to 1.

Our algorithm supports multiple sum constraints. As a first step we clip all values to the \([0, 1]\) range. We then iteratively project to respect all sum constraints. To achieve this we use a known algorithm [Wang and Carreira-Perpiñán, 2013; Duchi et al., 2008] to project onto a scaled probability simplex \(\sum a_i \leq B\) for a given bound \(B\). The algorithm repeatedly subtracts the surplus amount from all non-zero entries, clipping at 0. For example, given \(\{1.2, 1.0, 0.9, 0.5, 0.1\}\) and \(B = 2\), we subtract \((3.7 - 2)/5 = 0.34\) from each entry (clipping the last) to get \(\{0.86, 0.66, 0.56, 0.16, 0\}\) and then subtract \(0.24/4 = 0.06\) to get \(\{0.8, 0.6, 0.5, 0.1, 0\}\).

**Optimizing step size for gradient update:** Gradient ascent often works well with a constant step size. However, in some problems, when the rollout policy is random, the effect of the first action on \(Q\) is very small implying that the gradient is very small and a fixed step size is not suitable. We therefore search for an appropriate step size. To mitigate the fact that we are using continuous search over the discrete space we use a brute force search instead of standard line search.

We first choose a range for \(\alpha\) with enough flexibility so that any \(\alpha \in [0, \max\text{Alpha}]\) won’t push the variables beyond \(\max\text{Var} + 1\) or below -1 where \(\max\text{Var}\) is the max value among the variables. We then perform a linear search using 10 evenly spaced values \(a_0, a_1, \ldots, a_9\) in that region to find the best \(\alpha\) where quality is measured by the value of the \(Q\) function DAG. If the best \(\alpha\) happens to be \(a_0\), implying that the original \(\max\text{Alpha}\) may not be at the right scale, we reset the legal region to be \([0, a_0]\) and run the process again. This is repeated until the best \(\alpha\) is not \(a_0\) with a max of 5 levels.

**Sampling concrete Actions:** The gradient optimization performs a continuous search over the discrete space. This means that the values given by the \(Q\) are not always reliable on the fractional aggregate actions. To add robustness we associate each aggregate action encountered in the search with a concrete action chosen from its distribution and record the more reliable value of the concrete action. Search proceeds as usual with the aggregate actions but final action selection is done using these more reliable scores for concrete actions.

Selecting a concrete action from the aggregate one can be formulated as a weighted max SAT problem similar to [Sang et al., 2005]. However, this is an expensive calculation. We therefore use a heuristic that picks action variables with highest marginal probabilities as follows. We first sort action variables by their marginal probabilities. We then add active action bits as long as the marginal probability is not lower than marginal probability of random rollout and the constraints are not violated. For example, suppose the marginals are \(\{0.8, 0.6, 0.5, 0.1, 0\}\), \(B = 3\), and we use a threshold of 0.55. Then we have \(\{a_1, a_2\}\) as the final action.

**Stopping Criterion:** Our results show that gradient information is useful. However, getting precise values for the marginals at local optima is not needed, because small changes are not likely to affect the choice of concrete action. We thus use a loose criterion aiming to allow for a few gradient steps but to quit relatively early so as to allow for multiple restarts. Our implementation stops the gradient search if the max change in probability is less than \(\max\text{New} - \max\text{Old}\) \(\leq 0.1\). The result is an algorithm that combines Monte Carlo search and gradient based search.

### 4 Experimental Evaluation

We evaluate the SOGBOFA algorithm on several domains showing its advantage and in addition present an ablation study that shows that different components of the algorithm contribute to this success. The evaluation is done over 6 domains, elevators from the RDDL distribution, two from the International Probabilistic Planning Competition (IPPC) 2011, and three from IPPC 2014 where we chose domains amenable to expansion of the action space. We follow the setting of [Cui et al., 2015] and use the 10 competition problems (index 1 to 10) but add 10 more challenging instances in each domain (index 11 to 20). The size of the action space in these problems is shown in Table 1. The reported results represent averages from 20 runs in each problem instance where each instance is run for 40 time steps. In these runs each algorithm is given 60 seconds per step (on a cluster node having Intel Xeon X5675@ 3GHz CPU, and 24GB memory).

As baselines we compare against Rollout and MCTS that work by sampling concrete trajectories and against ARollout. As in [Cui et al., 2015], these algorithms simulate trajectories for \(\min\{20, \text{horizon}\}\) steps, which we denote as HalfDepth below. The work of [Cui et al., 2015] showed that on large factored spaces ARollout dominates PROST [Keller and Helmert, 2013] the winner of the recent IPPC so the comparison to ARollout serves as a reference point to state of the art. For visual clarity we rescale reward values relative to the values achieved by the random policy (as zero) and the value achieved by a simple hand coded policy for the domain (as 1) or against the value of one of the algorithms (which is the reference value of 1 in that case).

In addition we test several variants of the algorithm with some of the features disabled to investigate their contribution. In particular SARollout (Symbolic ARollout) uses the DAG
than the dynamic
\( \alpha \) many more actions, in some cases more than 10 fold more, seen by SARollout and the gradient algorithms. When in-

The top two rows in Figure 2 show a comparison of SOGBOFA to the baselines in all 6 domains showing a significant advantage on the problems with large action spaces.

The third row of Figure 2 shows a comparison to SARollout where for clarity in the comparison SARollout uses HalfDepth, exactly as Rollout. In addition these plots compare to HalfDepth variants of the gradient based algorithm with fixed \( \alpha = 0.1 \) vs. the dynamic setting. Due to space constraints we only show results for 3 of the domains but the other 3 domains show similar behavior. The plots show that SARollout dominates ARollout thus illustrating the significance of the symbolic representation even without gradients. They also show that the gradient based algorithms with HalfDepth either dominate SARollout or are comparable.

The choice between fixed and dynamic \( \alpha \) represents a tradeoff. They mostly perform pretty closely but in some cases one of the two dominates significantly. The importance of the dynamic selection of \( \alpha \) arises in some of the large problems (e.g., academic advising 16-20) where the gradient with respect to action variables is very small (around \(-10^{-16}\)), and fixing the alpha to a small value makes for tiny updates that make no progress. We prefer the dynamic setting as it can adapt to the shape of the \( Q \) function more robustly.

It is insightful to look at the number of concrete actions seen by SARollout and the gradient algorithms. When inspected (details omitted) we observe that SARollout sees many more actions, in some cases more than 10 fold more, than the dynamic \( \alpha \) variant. Nonetheless, the gradient algorithms that see less actions have better performance.

The fourth row of Figure 2 evaluates the contribution of dynamic depth selection for 3 domains. Results for the other domains are omitted due to space constraints and they show similar trends. We can observe that dynamic depth is comparable and in some cases significantly better than fixed depth.

The left plot in the fifth row of Figure 2 illustrates that projection is crucial on the sysadmin domain. In this case, without projection the algorithm is not distinguishable from the random policy. The remaining plots in the fifth row illustrate the importance of the tradeoff given by the stopping criterion. Results are shown as a function of the \( L_1 \) bound \( S \) (\( S=0.1 \) in the main algorithm). The plots show results for \( S \) in the range \( 10^{-9} \) (very strict criterion, few restarts) to 1 (always stop after one gradient update) for problem 20 of sysadmin and academic advising. For reference we also include the SARollout algorithm which makes no updates, and this is represented in the plot at the point \( S = 2 \). The results show that neither extreme is a good choice, that a criterion balancing gradients and Monte Carlo steps provides better performance and that there is a large range of values that provide this tradeoff.

5 Conclusion

The paper presented a novel algorithm for stochastic planning in factored spaces that scales well to very large action spaces. The main idea is to use symbolic simulation with an explicit approximation induced by independence assumptions, and to perform gradient search to maximize the \( Q \) function using this symbolic representation. Several algorithmic tools serve to make the algorithm robust, including dynamic choice of simulation depth, dynamic choice of gradient step size, shadowing the gradient search with a search over concrete actions, and a balance between gradients and random restarts. The experiments demonstrate that the algorithm is significantly better than previous work and that the different components of the algorithm contribute to that success.

The algorithm also suggests an intriguing opportunity going beyond symbolic rollouts. In particular, as described above, when constructing the DAG we can leave all action variables in their symbolic form. In that case the \( Q \) function is a function of all action variables in all time steps. Selecting concrete values for all these variables is the same as deciding on the entire sequence of actions in advance of execution. This is known as conformant planning [Palacios and Geffner, 2009; Domshlak and Hoffmann, 2006; Lee et al., 2014]. Interestingly, reverse accumulation automatic differentiation allows us to perform gradient steps over all these variables without an increase in run time. We can therefore use essentially the same method to optimize an approximation of conformant planning. This has the potential to improve over SOGBOFA, especially when rolling out the random policy is not informative. Preliminary experiments suggest that such improvement is obtained in some cases but that in many cases the large scale optimization leads to degradation in performance. We leave exploration of this optimization challenge for future work.

Acknowledgments

We are grateful to Alan Fern, Prasad Tadepalli and Scott Sanner for helpful discussions. This work was partly supported by NSF under grant IIS-0964457. Some of the experiments in this paper were performed on the Tufts Linux Research Cluster supported by Tufts Technology Services.
Figure 2: Data from experimental evaluation. The top two rows compare SOGBOFA to the baselines showing significant advantage on problems with large action spaces (on the right of the plots). The third row, where all algorithms including SOGBOFA are fixed at HalfDepth, compares Symbolic ARollout to SOGBOFA with and without step size optimization. The forth row compares SOGBOFA to the variant fixed at HalfDepth. The fifth row (left) shows the necessity of projection, and (middle, right) non-sensitivity with respect to the $L_1$ bound parameter in the stopping criterion.
References


Appendix

A Additional Results

This section includes additional experimental results that were omitted from the main paper due to space constraints.

For SARollout comparison: Table 2 shows the number of actions sampled from each algorithm. The results are consistent among all domains, but we find that SARollout and the two fixed depth gradient algorithms, cannot build the tree to half horizon on tamarisk problems 14-20 given 60 seconds. Run logs show that they only can build the tree to depth 2-9. In addition, the dynamic depth variant only builds the DAG to depth 2-7 on these problems. This illustrates the benefit of dynamic depth, which works much better on these problems.
Table 2: Number of action sampled. From top to bottom: academic_advising, tamarisk, skill_teaching, sysadmin, traffic, elevators

<table>
<thead>
<tr>
<th>Problem</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA Rollout</td>
<td>11</td>
<td>36</td>
<td>16</td>
<td>121</td>
<td>21</td>
<td>211</td>
<td>26</td>
<td>326</td>
<td>31</td>
<td>466</td>
<td>5712</td>
<td>8299</td>
<td>6437</td>
<td>4419</td>
<td>4011</td>
<td>2685</td>
<td>2194</td>
<td>2158</td>
<td>1878</td>
<td>1603</td>
</tr>
<tr>
<td>Dynamic Alpha</td>
<td>11.0</td>
<td>36</td>
<td>16</td>
<td>121</td>
<td>21</td>
<td>211</td>
<td>26</td>
<td>326</td>
<td>31</td>
<td>466</td>
<td>5712</td>
<td>8299</td>
<td>6437</td>
<td>4419</td>
<td>4011</td>
<td>2685</td>
<td>2194</td>
<td>2158</td>
<td>1878</td>
<td>1603</td>
</tr>
<tr>
<td>Fix Alpha</td>
<td>11</td>
<td>36</td>
<td>16</td>
<td>121</td>
<td>21</td>
<td>211</td>
<td>26</td>
<td>326</td>
<td>31</td>
<td>466</td>
<td>5712</td>
<td>8299</td>
<td>6437</td>
<td>4419</td>
<td>4011</td>
<td>2685</td>
<td>2194</td>
<td>2158</td>
<td>1878</td>
<td>1603</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA Rollout</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>17</td>
<td>211</td>
<td>3142</td>
<td>23909</td>
<td>24008</td>
<td>2417</td>
<td>2373</td>
<td>23884</td>
<td>24565</td>
<td>2397</td>
<td>1978</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic Alpha</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>17</td>
<td>211</td>
<td>3142</td>
<td>23909</td>
<td>24008</td>
<td>2417</td>
<td>2373</td>
<td>23884</td>
<td>24565</td>
<td>2397</td>
<td>1978</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fix Alpha</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>15</td>
<td>15</td>
<td>17</td>
<td>17</td>
<td>211</td>
<td>3142</td>
<td>23909</td>
<td>24008</td>
<td>2417</td>
<td>2373</td>
<td>23884</td>
<td>24565</td>
<td>2397</td>
<td>1978</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Number of legal actions in test problems

Figure 3: Comparison of SOGBOFA to ARollout and other baselines.
Figure 4: Comparison between ARollout, SARollout, and variants of gradient with fixed depth.

Figure 5: Comparison between dynamic depth and fixed depth.

Figure 6: Evaluation of projection (left) and comparison when varying the stopping criterion for gradient search.
<table>
<thead>
<tr>
<th>criterion</th>
<th>0.000000001</th>
<th>0.0000001</th>
<th>0.001</th>
<th>0.01</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>restarts</td>
<td>197.85</td>
<td>1740.67</td>
<td>60.8</td>
<td>59.523</td>
<td>51.05</td>
<td>47.0</td>
<td>20.05</td>
<td>20.705</td>
<td>20.105</td>
<td>19.36</td>
</tr>
<tr>
<td>updates</td>
<td>197.85</td>
<td>1740.67</td>
<td>60.8</td>
<td>59.523</td>
<td>51.05</td>
<td>47.0</td>
<td>20.05</td>
<td>20.705</td>
<td>20.105</td>
<td>19.36</td>
</tr>
<tr>
<td>actions</td>
<td>1740.67</td>
<td>197.85</td>
<td>60.8</td>
<td>59.523</td>
<td>51.05</td>
<td>47.0</td>
<td>20.05</td>
<td>20.705</td>
<td>20.105</td>
<td>19.36</td>
</tr>
<tr>
<td>criterion</td>
<td>0.000000001</td>
<td>0.0000001</td>
<td>0.001</td>
<td>0.01</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>restarts</td>
<td>232.57</td>
<td>1603.61</td>
<td>118.23</td>
<td>104.115</td>
<td>85.16</td>
<td>54.8</td>
<td>48.135</td>
<td>20.06</td>
<td>8.905</td>
<td>5.735</td>
</tr>
<tr>
<td>updates</td>
<td>232.57</td>
<td>1603.61</td>
<td>118.23</td>
<td>104.115</td>
<td>85.16</td>
<td>54.8</td>
<td>48.135</td>
<td>20.06</td>
<td>8.905</td>
<td>5.735</td>
</tr>
</tbody>
</table>

Table 4: statistics of different stopping criterion. From top to bottom: academic_advising, sysadmin