28th International Conference on
Automated Planning and Scheduling

June 24–29, 2018, Delft, the Netherlands

IPC 2018 – Probabilistic Track

Planner Abstracts for the

Probabilistic Track

in the

International Planning Competition 2018

Edited by:

Thomas Keller.

Organization

Thomas Keller
University of Basel, Switzerland

Contents

The SOGBOFA system in IPC 2018: Lifted BP for Conformant Approximation of Stochastic Planning

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

Abstract

The SOGBOFA algorithm estimates the value of an action for the current state by building an explicit computation graph capturing an approximation of the value obtained when starting with this action and continuing with a random policy. This is combined with automatic differentiation over the graph to search for the best action. This approach was shown to be competitive in large scale planning problems with factored state and action spaces. The systems submitted to the International Planning Competition (IPC) introduce two improvements of SOGBOFA. The first improvement builds on the recently observed connection between SOGBOFA and belief propagation to improve efficiency by lifting its computation graph, taking inspiration from lifted belief propagation. The second improves the rollout policy which is used in the approximate computation graph. Instead of rolling out a trajectory of the random policy, the trajectory actions are optimized at the same time as the initial action. The two variants submitted to the IPC include both improvements but differ in how trajectory actions are optimized. In addition, due to changes in the specification language for the IPC, new facilities for handling action constraints were incorporated in the system.

Introduction

This paper gives an overview of variants of the SOGBOFA system that participated in the probabilistic track of the international Planning Competition (IPC), 2018. SOGBOFA (Cui and Khardon 2016) extends the well known rollout algorithm (Tesauro and Galperin 1996). The rollout algorithm uses a simulator to estimate the quality of each possible action for the first step by taking that action and then continuing the simulation with some fixed policy. Multiple simulations are required for each fixed action to get a reliable estimate. But once this is done one can perform policy improvement or just use the best action for the current state. SOGBOFA improves over this algorithm in two important ways. The first is that instead of using concrete simulation of trajectories the algorithm builds an explicit computation graph capturing an approximation of the corresponding value when rolling out the random policy. Therefore a single symbolic simulation suffices. The second is that because the simulation is given in an explicit computation graph one can use automatic differentiation and gradients to search for the best action, avoiding the action enumeration which is required by rollout. This approach was shown to be competitive in large scale planning problems with factored state and action spaces where such enumeration is not feasible.

Two improvements of the SOGBOFA system were added for the competition. In recent work we have shown that the computation graph of SOGBOFA calculates exactly the same solution as the one that would be computed by belief propagation (BP) on the corresponding inference problem (Cui and Khardon 2018; Cui, Marinescu, and Khardon 2018). The first improvement uses a Lifted version of SOGBOFA, taking inspiration from lifted belief propagation. The idea in lifted BP is to avoid repeated messages during computation and calculate the result in aggregate. For SOGBOFA this turns out to be a simple modification of the construction of the computation graph.

The second improvement uses conformant approximation. The quality of the approximation of SOGBOFA is limited by the fact that it uses a random policy for rollout. For some domains this provides enough information to distinguish the best first action but for others this does not work. The conformant approximation learns a fixed sequence of actions to be used for rollout from the current state (this is similar to the plan used in conformant planning, hence the name for this approximation). The choice of these actions is optimized simultaneously with the optimization of the first action, using the same computation graph and gradient computation.

The two systems submitted to the competition use both improvements, using the lifted graph representation and optimizing all action variables simultaneously. The difference between the two is in how trajectory actions are optimized. The first system uses fractional values for trajectory action variables during search whereas the second system projects them to binary values before evaluation.

IPC 2018 has modified the RDDL (Sanner 2010) specification of domains by moving action preconditions into a separate constraints section and adding several other types of action constraints in that section that must be handled by the planner. The SOGBOFA entries for the IPC extend the original system to handle these constructs.

The rest of the paper is structured as follows. The next

section gives an overview of the original SOGBOFA algorithms. The following 3 sections describe lifting, the conformant approximation and constraints handling. The final section briefly discusses competition results.

The Basic SOGBOFA Algorithm

Stochastic planning can be formalized using Markov decision processes (Puterman 1994) in factored state and action spaces. In factored spaces (Boutilier, Dean, and Hanks 1995) the state is specified by a set of variables and the number of states is exponential in the number of variables. Similarly in factored action spaces an action is specified by a set of variables. We assume that all state and action variables are binary. Finite horizon planning can be captured using a dynamic Bayesian network (DBN) where state and action variables at each time step are represented explicitly and the CPTs of variables are given by the transition probabilities. In off-line planning the task is to compute a policy that optimizes the long term reward. In contrast, in on-line planning we are given a fixed limited time, t seconds, per step and cannot compute a policy in advance. Instead, given the current state, the algorithm must decide on the next action within t seconds. Then the action is performed, a transition and reward are observed and the algorithm is presented with the next state. This process repeats and the long term performance of the algorithm is evaluated.

AROLLOUT and SOGBOFA perform on-line planning by estimating the value of initial actions where a fixed rollout policy, typically a random policy, is used in future steps. The AROLLOUT algorithm (Cui et al. 2015) introduced the idea of algebraic simulation to estimate values but optimized over actions by enumeration. Then Cui and Khardon (2016) showed how algebraic rollouts can be computed symbolically and that the optimization can be done using automatic differentiation. The high level structure of SOGBOFA is:

SOGBOFA(S)

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

Overview of the Algorihm: 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: Finite horizon planning can be translated from a high level language (e.g., RDDL (Sanner 2010)) to a dynamic Bayesian network (DBN). AROLLOUT transforms the CPT of a node x into a disjoint sum form. In particular, the CPT is represented in the form $\text{if}(c_{11}|c_{12}...)\ then\ p_1\ ...\ \text{if}(c_{n1}|c_{n2}...)\ then\ p_n$, where p_i is $p(x{=}1)$ and the c_{ij} are conjunctions of parent values which are are mutually exclusive and exhaustive. It then performs a forward pass calculating $\hat{p}(x)$, an approximation of the true marginal $p(x)$, for any node x in the graph. $\hat{p}(x)$ is calculated as a function of $\hat{p}(c_{ij})$, an estimate of the probability that c_{ij} is true, which assumes the parents are independent. This is done using the following equations where nodes are processed in the topological order of the graph:

$$\hat{p}(x) = \sum_{ij} p(x|c_{ij})\hat{p}(c_{ij}) = \sum_{ij} p_i \hat{p}(c_{ij}) \quad (1)$$

$$\hat{p}(c_{ij}) = \prod_{w_k \in c_{ij}} \hat{p}(w_k) \prod_{\bar{w}_k \in c_{ij}} (1 - \hat{p}(w_k)). \quad (2)$$

The following example from (Cui and Khardon 2016) illustrates AROLLOUT and SOGBOFA. 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 transitions and reward are given by the following RDDL (Sanner 2010) expressions where primed variants of variables represent the value of the variable after performing the action.

```
cond1 = Bernoulli(0.7)
cond2 = Bernoulli(0.5)
s'(1) = if (cond1)  then ~a(3)  else false
s'(2) = if (s(1))   then a(2)   else false
s'(3) = if (cond2)  then s(2)   else false
reward = s(1) + s(2) + s(3)
```

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

```
s'(1) = (1-a(3))*0.7
s'(2) = s(1)*a(2)
s'(3) = s(2) * 0.5
r = s(1) + s(2) + s(3)
```

These expressions are used to calculate an approximation of marginal distributions over state and reward variables. The distribution at each time step is approximated using a product distribution over the state variables. To illustrate, assume that the state is $s_0 = \{s(1){=}0, s(2){=}1, s(3){=}0\}$ which we take to be a product of marginals. At the first step AROLLOUT uses a concrete action, for example $a_0 = \{a(1){=}1, a(2){=}0, a(3){=}0\}$. This gives values for the reward $r_0 = 0 + 1 + 0 = 1$ and state variables $s_1 = \{s(1){=}(1-0)*0.7{=}0.7, s(2){=}0*0{=}0, s(3){=}1{*}0.5 = 0.5\}$.

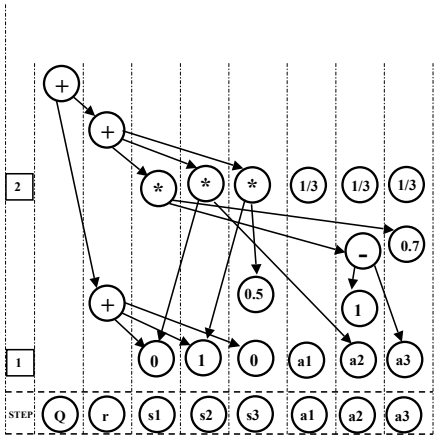

Figure 1: Example of SOGBOFA graph construction.

In future steps it calculates marginals for the action variables and uses them in a similar manner. For example if $a_1 = \{a(1)$=0.33, $a(2)$=0.33, $a(3)$=0.33$\}$ we get $r_1 = 0.7 + 0 + 0.5 = 1.2$ and $s_2 = \{s(1)$=$(1 - 0.33) * 0.7$, s(2)= 0.7 * 0.33, $s(3)$=$0 * 0.5\}$. Summing the rewards from all steps gives an estimate of the Q value for a_0. AROLLOUT randomly enumerates values for a_0 and then selects the one with the highest estimate.

The main observation in SOGBOFA is that instead of calculating concrete values, we can use the expressions to construct an explicit directed acyclic graph representing the computation steps, where the last node represents the expectation of the cumulative reward. SOGBOFA uses a symbolic representation for the first action and assumes that the rollout uses the random policy. In our example if the action variables are mutually exclusive (such constraints are used imposed in high level domain descriptions) this gives marginals of $a_1 = \{a(1)$=0.33, $a(2)$=0.33, $a(3)$=0.33$\}$ over these variables. The SOGBOFA graph for our example expanded to depth 1 is shown In Figure 1. The bottom layer represents the current state and action variables. Each node at the next level represents the expression that AROLLOUT would have calculated for that marginal. To expand the planning horizon we simply duplicate the second layer construction multiple times.

Now, given concrete marginal for the action variables at the first step, i.e., a_0, an evaluation of the graph captures the same calculation as AROLLOUT.

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 $Q^\pi(s, a)$ and gradient computation are expensive so that the number of actions explored in the search might be too small. SOGBOFA first estimate the time cost for calculating gradients and performing updates *per node in the graph*. It then estimates the potential graph size as a function of simulation depth. The rollout depth is chosen to guarantee at least 200 updates on action marginals.

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). 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.

Maintaining action constraints: Gradient updates allow the values of marginal probabilities to violate the $[0, 1]$ range constraint on marginals as well as explicit constraints on legal actions. The original version of SOGBOFA only allowed for sum constraints of the form $\sum a_i \leq B$. 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 or in cases of limited concurrency.

To handle this issue we use projected gradient ascent (Shalev-Shwartz 2012), where parameters are projected into the legal region after each update. The projection step uses an iterative procedure from (Wang and Carreira-Perpiñán 2013; Duchi et al. 2008) that supports constraints of the form $\sum a_i \leq B$. The algorithm repeatedly subtracts the surplus amount ($(\sum a_i - B)/k$ for k action variables) from all non-zero entries, clipping at 0, until the surplus is 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 the Q value is very small implying that the gradient is very small and a fixed step size is not suitable. This also implies that we need to search over a large range for an appropriate step size. SOGBOFA performs this search hierarchically. We start with a large fixed range and search over a grid set of values. Then if the value chosen is the smallest one tested we recurse to a smaller range.

Sampling concrete Actions: The search assigns numerical marginal probabilities for each action variable. We need to select a concrete action from this numerical representation. In addition the selected action must satisfy the action constraints as mentioned above. SOGBOFA uses a greedy heuristic 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. This procedure is used for selecting the final action to use during online planning.

In addition to the above, we also use action selection during the search (in step 9 of the algorithm). The gradient optimization performs a continuous search over the discrete space. This means that the values given by the graph 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 as in the previous paragraph 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.

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 the 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 $S = 0.1$, that is, $\|A_{new} - A_{old}\|_1 \leq 0.1$. The result is an algorithm that combines Monte Carlo search and gradient based search.

Lifted SOGBOFA

In recent work (Cui and Khardon 2018; Cui, Marinescu, and Khardon 2018) we showed that the approximate value computed by SOGBOFA is identical to the value that would be computed by belief propagation. Motivated by that we proposed a lifted planning algorithm that takes inspiration from lifted BP (Singla and Domingos 2008; Kersting, Ahmadi, and Natarajan 2009). In Lifted BP, two nodes or factors send identical messages when all their input messages are identical and in addition the local factor is identical. With the computational structure of SOGBOFA this has a clear analogy. Two nodes in SOGBOFA's graph are identical when their parents are identical and the local algebraic expressions capturing the local factors are identical. This suggests a straightforward implementation for Lifted SOGBOFA using dynamic programming.

The lifted algorithm is as follows. We run the algorithm in exactly the same manner as SOGBOFA except for the construction of the computation graph. (1) When constructing a node in the explicit computation graph we check if an identical node, with same parents and same operation, has been constructed before. If so we return that node. Otherwise we create a new node in the graph. (2) If we only use the idea above we may end up with multiple identical edges between the same two nodes. Such structures are simplified during construction. In particular, every `plus` node with k incoming identical edges is turned into a `multiply node` with constant multiplier k. Similarly, every `multiply` node with k incoming identical edges is turned into a `power node` with constant power k. This automatically generates the counting expressions that are often seen in lifted inference.

Conformant SOGBOFA

SOGBOFA uses a random policy for trajectory actions, that is, for actions taken after the first step. In some domains the value achieved by the random policy is already indicative of the value of the state it is started from. In these cases SOGBOFA is successful. In some domains a random policy masks any information and one must use a more informed type of lookahead. For SOGBOFA there is a natural way to form this idea: one can try to improve the rollout policy. It turns out, however, that optimizing a policy within the time to select an action for online planning is not realistic. In addition, because we only maintain aggregate marginals for states after the first step, the simulation cannot condition the policy on a concrete state so using a policy is not necessarily beneficial. We therefore directly optimize the marginals of the variables instead of optimizing a policy representation. In other words, the "rollout policy" is a fixed sequence of actions which is optimized during planning. This type of solution is often called *conformant planning*. Recall that for the first action we use projection to binary values to improve the search. We can use the same process for trajectory actions. This is the the variant CONFORMANT-SOGBOFA-B-IPC18 in the competition. Alternatively, we can keep the trajectory actions as fractional values. This is similar to the use of the random policy, and intuitively it makes more sense for trajectory actions because we are not interested in selecting a concrete value for them, but instead interested in getting an aggregate simulation. This is the the variant CONFORMANT-SOGBOFA-F-IPC18 in the competition. Either way, the process of evaluating the first action also chooses a conformant plan that best supports that first action. The conformant plan is only used for the purpose of action evaluation. It is not used for controlling the MDP. Instead, as always done in online planning, the process restarts its computation after the first action has been taken.

The SOGBOFA graph immediately supports this type of optimization because trajectory action variables are represented as nodes in the graph. Instead of assigning these nodes numerical values imposed by the random policy we retain them as symbolic variables and optimize them along with the first action. Note that reverse mode automatic differentiation supports calculation of gradients w.r.t. all nodes with the same time complexity so that there is no significant change to run time. However, in preliminary experiments we have found that optimizing a large number of action variables from all time steps requires significantly more gradient steps to reach a good solution.

Conformant SOGBOFA adjusts for this by modifying the dynamic depth selection heuristic. As before, we estimate the cost of gradient computation *per node* in the graph and the expected graph size as a function of depth. Our heuristic here is to select the depth so as to guarantee $200 * 2^i$ updates, where i is the conformant search depth.

Note that in principle we could separate search depth from conformant depth, for example, optimize actions for the first few steps and thereafter use the random policy. This can allow for a more refined control of the time tradeoff for the search. However, for the IPC we did not implement such a scheme. In our submission, we always use the same dynamic depth and conformant depth.

Handling Constraints

Previous versions of the probabilistic IPC integrated action preconditions into the transition function. With this, if an action is applied in a state where it is not legal it is simply a no-op and therefore not useful. Action constraints were limited to the sum constraints discussed above. The current IPC represents action preconditions as constraints, and if an illegal action is attempted the planner fails. In addition, the types

of constraints were expanded to include more constraints on actions.

Our implementation parses the action constraints and handles each type in a separate manner, following the approach in the previous treatment in SOGBOFA.

First, action preconditions have the form: "for all arguments x, if $a(x)$ is used, then $c(x)$ must be true", where $c(x)$ is the precondition. When we identify this form, we embed it into the transition function to mimic the previous IPC encoding. In particular, consider a concrete action bit $a(o)$ for some concrete objects o. We replace each occurrence of $a(o)$ in the transition function by $a(o) \wedge c(o)$. In other words, during simulation we assume that illegal actions are no-op.

Second, constraints of the form $\sum a_i \leq B$ are treated exactly as before.

The IPC introduced additional types of constraints. The first includes conditions of the form: "some quantifiers x, condition(x) $\rightarrow$ some quantifiers y, actions(x, y)" where actions(x, y) is a subset of the ground action variables, and the quantifiers over action arguments and other objects can be existential or universal as appropriate. When actions(x, y) include just one action then we have a forced action. When actions(x, y) represents a disjunction of action variables, this means that at least one of these actions must be chosen. A similar situation arises with $\sum a_i \geq 1$ which captures a disjunction and with $\sum a_i = 1$ which has both the upper bound and a disjunction. It should be clear the SOGBOFA is not well matched for handling general action constraints. Our implementation uses the previous methodology to support the new constraints as follows.

For constraints with forced actions we can evaluate the condition to a value v. In a concrete state the condition evaluates to 0 or 1 and in an aggregate state it evaluates to (an approximation of) the probability that the condition holds. We then replace the marginal probability p for for the corresponding a_i by $p \leftarrow \max\{p, v\}$. Note that if $v = 0$ then p does not change and if $v = 1$ then $p = 1$ which means that action selection algorithm described above will pick this action variable first, so we comply with the forced action constraint. In an aggregate state we potentially increase p to be as high as the probability that the condition holds v. Note that this implementation supports the conformant algorithm in the same manner as the action of the first step so no distinction is needed.

Our implementation for an implied disjunction of action variables, and for an implication with $\sum a_i \geq 1$ on the right hand side is similar. We first evaluate the condition to a value v. We then pick the a_i with the highest marginal probability p among the ones in the disjunction and replace it with $\max\{p, v\}$. As with forced actions, if the condition is true then we force at least one of the relevant action variables to be true as well. With aggregate states we get a correction to the marginal probabilities on action variables.

Finally we give a separate treatment to constraints where the condition is always true and the outcome is a disjunction. For example this includes $\sum a_i = 1$ without an associated condition. If we used the implementation of the previous paragraph in aggregate states this will force at least one action variable to be 1 and if the constraint is $\sum a_i = 1$ the trajectory actions in conformant SOGBOFA will always use discrete 0,1 values. This will hinder the search that uses marginal probabilities and gradients which is the main advantage of our method. Instead, for this type of constraint we first calculate $\sum p_i$ where p_i is the current marginal probability of a_i. If $\sum p_i > 1$ we use projection as explained above. If $\sum p_i < 1$ we add $1 - \sum p_i$ to the largest among the p_i. In this way the constraint is satisfied on the fractional values but we do not force any specific action variable among trajectory actions to 1 during the search.

Summary

This paper describes variants of the SOGBOFA system that participated in the IPC 2018. Two algorithmic extensions were developed including lifting the computation for speedup and the use of the conformant heuristic to improve the quality of the search. The new form of action preconditions and action constraints included in the IPC required extensions to handle forced actions and an implied disjunction of possible actions. These are naturally implemented within the current system by modifying calculated marginal probabilities to enforce a logical or probabilistic form of the constraints.

Notes on Competition Results

CONFORMANT-SOGBOFA-F-IPC18 was runner up in the competition. This is despite failures on several of the domains which are discussed next.

The system failed (zero score) in the Wildlife Preserve domain. This is due to not supporting Enumerable variables. The No-Enumerable translated domain is large and the RDDL simulator which is embedded in our system is stuck while processing the file so that the planner does not get to start solving the problems.

The system failed (zero score) in the Manufacturer domain. This is due to having an unsupported constraint of the form $\sum a_i \leq \sum s_j$ where a_i are action variables and s_j are state variables. This can be supported in the same manner as other constraints but was not implemented for the competition.

The system also failed on many instances in Chromatic Dice and Push Your Luck due to incorrect enforcement of constraints in boundary cases which reduced its overall score for these domains.

Bug fixes to these issues will be submitted to the IPC repository.

In summary, like SOGBOFA, Lifted Conformant SOGBOFA has an inherent limitation due to the aggregate simulation that produces approximate values. However, it provides a good tradeoff between accuracy and performance when the problems are large and combinatorially challenging so that other solvers must approximate as well. Forward aggregate simulation does not interact well with action constraints and this was a significant challenge in this competition.

Acknowledgments

This work was partly supported by NSF under grant IIS-1616280.

References

Boutilier, C.; Dean, T.; and Hanks, S. 1995. Planning under uncertainty: Structural assumptions and computational leverage. In *Proceedings of the Second European Workshop on Planning*, 157–171.

Cui, H., and Khardon, R. 2016. Online symbolic gradient-based optimization for factored action MDPs. In *Proc. of the International Joint Conference on Artificial Intelligence*.

Cui, H., and Khardon, R. 2018. Stochastic planning, lifted inference, and marginal MAP. In *Workshop on Planning and Inference help with AAAI*.

Cui, H.; Khardon, R.; Fern, A.; and Tadepalli, P. 2015. Factored MCTS for large scale stochastic planning. In *Proc. of the AAAI Conference on Artificial Intelligence*.

Cui, H.; Marinescu, R.; and Khardon, R. 2018. From stochastic planning to marginal MAP. In *Advances in Neural Information Processing Systems*.

Duchi, J. C.; Shalev-Shwartz, S.; Singer, Y.; and Chandra, T. 2008. Efficient projections onto the l_1-ball for learning in high dimensions. In *Proceedings of the International Conference on Machine Learning*, 272–279.

Griewank, A., and Walther, A. 2008. *Evaluating derivatives - principles and techniques of algorithmic differentiation (2. ed.)*. SIAM.

Kersting, K.; Ahmadi, B.; and Natarajan, S. 2009. Counting belief propagation. In *UAI*, 277–284.

Puterman, M. L. 1994. *Markov decision processes: Discrete stochastic dynamic programming*. Wiley.

Sanner, S. 2010. Relational dynamic influence diagram language (rddl): Language description. *Unpublished Manuscript. Australian National University*.

Shalev-Shwartz, S. 2012. Online learning and online convex optimization. *Foundations and Trends in Machine Learning* 4(2):107–194.

Singla, P., and Domingos, P. M. 2008. Lifted first-order belief propagation. In *AAAI*.

Tesauro, G., and Galperin, G. R. 1996. On-line policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems*, 1068–1074.

Wang, W., and Carreira-Perpiñán, M. Á. 2013. Projection onto the probability simplex: An efficient algorithm with a simple proof, and an application. *CoRR/arXiv* abs/1309.1541.

Imitation-Net: A Supervised Learning Planner

Alan Fern, Murugeswari Issakkimuthu and Prasad Tadepalli
School of EECS, Oregon State University
Corvallis, OR 97331, USA

Abstract

Imitation-Net is an offline planner based on supervised learning to imitate an expert policy for the problem. It works in two phases - Training and Evaluation. During the training phase it creates a training dataset containing trajectories from an expert policy and trains a deep neural network (Goodfellow, Bengio, and Courville 2016) to approximate that policy. During the evaluation phase it runs the trained network to get an action for the current state by means of a single feed-forward pass.

Introduction

Figure 1 shows the schematic diagram of the entire planning system. There are three components: *RDDLSim* (Sanner 2010), the Java *RDDL* server used for evaluation in the competition, a C++ dynamic library based on *Prost* (Keller and Eyerich 2012), and a Python component consisting of a Tensor Flow policy network. *Prost* is the state-of-the-art search-based online planner for *RDDL* domains. The C++ dynamic library based on *Prost* acts as an intermediate layer between the *RDDL* server and the Tensor Flow network providing routines for communicating with the server, running the evaluation loop and simulating trajectories during the training phase.

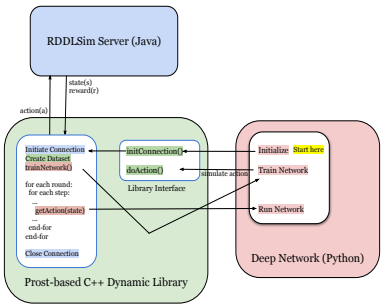

Figure 1: Schematic Diagram

The Python component starts the control flow by calling *initConnection()* in the dynamic library sending handles of callback functions *train()* and *test()* for training and running the policy network respectively. The dynamic library actually initiates (and also terminates) the communication with the server, receives and parses the *RDDL* domain and problem files, initializes the required data structures, and starts the network training process by invoking the *train()* callback function. The nested *for* loops in the dynamic library denote the evaluation loop in which it returns an action for the current state to the server and receives the reward and next state from the server. At each planning step the library invokes the *test()* callback function to run the policy network with the current state s and returns the received action $\hat{a}$ to the server.

The Policy Network

Architecture. The policy network is a sparsely connected structure with three hidden layers. The input layer has as many nodes as the number of state-fluents (n) in the problem, the hidden layers have $C * n$ nodes, and the output layer has as many nodes as the number of action-fluents in the problem (m). The hidden layers have *ReLU* non-linear units. The output layer has *sigmoid* non-linear units for the action nodes, thereby supporting factored action spaces directly. The network is not fully connected except at the final layer and connections going into the hidden layers are customized for each problem based on the transition function for state-fluents in the *RDDL* description. The input nodes represent state-fluents and the nodes in the hidden layers are clustered into n groups of C nodes each with each group representing a state-fluent. A hidden node corresponding to a state-fluent receives connections only from its parent nodes (in the previous layer) according to the state-fluent transition dynamics very much resembling a Dynamic Bayesian Network *(DBN)* structure. Figure 2 shows a sparse network for a problem with 4 state-fluents and 5 action-fluents for $C = 1$ and figure 3 roughly shows the same network for $C = 5$. The parameter C is called *channels* in the same sense as in convolutional neural networks. Reference (Issakkimuthu, Fern, and Tadepalli 2018) has more details on this sparse architecture.

Training Data Generation As shown in the schematic diagram in figure 1 the procedure for training data generation is kept within the C++ dynamic library mainly because it uses

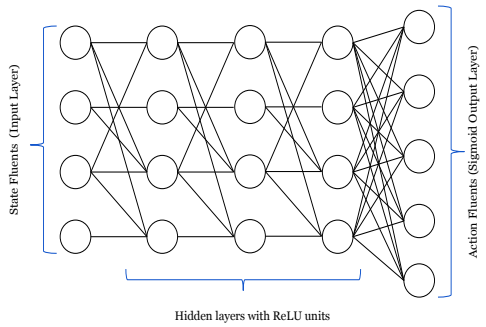

Figure 2: Network Architecture with one channel

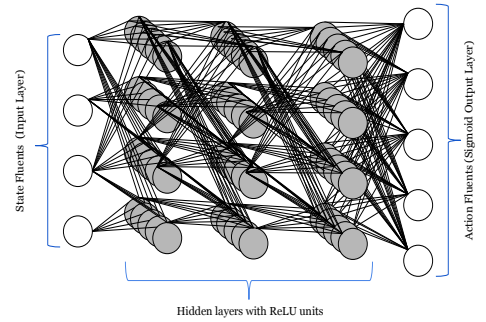

Figure 3: Network Architecture with 5 channels

some of the functionalities already available in *Prost*. The expert policy to imitate is the *Rollout-of-Random* policy, which is a one-step greedy policy over the value function of the random policy, i.e., the policy next in sequence to the random policy in the regular policy iteration sequence. Trajectories of the *Rollout-of-Random* policy are generated by estimating $Q^{\pi}(s, a)$ for each action $a \in A_s$ at state s for the random policy π and taking the best action to move to the next state s' from which the process is repeated. Therefore, states in the training set can be said to follow the state distribution of *Rollout-of-Random*.

Training The network is written in Tensor Flow and trained using Stochastic Gradient Descent *(SGD)* with a batch size of 10 to minimize the cross-entropy loss using the built-in *Adam* optimizer. During evaluation the probabilities of action-fluents computed by the trained network are used to compute the probabilities of individual applicable actions and the one with the highest probability is selected.

Implementation Details

The Python - C++ interface is implemented using the *ctypes* library (https://docs.python.org/3/library/ctypes.html). The important functionalities in *Prost* used in the dynamic library are

1. The *IPPCClient* class for establishing (and terminating) the connection with the *RDDL* server, parsing the *RDDL* domain and problem files and initializing data structures, and running the evaluation loop receiving state and reward signals and sending actions
2. The *RandomWalk* class for simulating a trajectory from state s starting with action a and then following the random policy π for h steps accounting for steps 12 through 19 in algorithm **??**
3. The *IDS* class to estimate the best rollout horizon h for the problem by means of iterative deepening search

Parameter Settings:

1. The competition imposes a *RAM* limit of 4GB for the planner. *RAM* usage is periodically monitored in the C++ function that creates the training dataset and the function is terminated once a limit of 2.5GB is reached. *RAM* usage is also monitored in the Python program while training the network and the training process is terminated once a limit of 3.5GB is reached.
2. The maximum number of training records in the dataset is limited to 30000, since larger datasets cannot be processed in the training process in limited time in the competition setting. The rollout horizon h for training data generation is initialized to the minimum of 5 or the value returned by the *IDS* class.
3. The total time (T) available to solve a problem instance needs to be divided between the training and evaluation phases leaving enough time for other associated computations like the initial parsing process. Approximately 70% of the total time T is set aside for just training the network. To be precise, an untrained network is run for one-fifth (15 for the competition) of the total number of episodes (75 in the competition) to compute a time t and time for final evaluation (t_e) is set to $2 \times t$ times the total number of episodes and time for data generation and training is set to 80% of $T - T_e$ out of which 30% is alloted for data generation and 70% is alloted for training.

Acknowledgements

Many thanks to Dr. Thomas Keller for his help with resolving problems connected to *Prost* functionalities.

References

[Goodfellow, Bengio, and Courville 2016] Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep learning*. MIT press.

[Issakkimuthu, Fern, and Tadepalli 2018] Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems.

[Keller and Eyerich 2012] Keller, T., and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

[Sanner 2010] Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language description.

Random-Bandit: An Online Planner

Alan Fern, Murugeswari Issakkimuthu and Prasad Tadepalli
School of EECS, Oregon State University
Corvallis, OR 97331, USA

Abstract

Random-Bandit is an online planner based on the ϵ-greedy algorithm for multi-armed bandit problems (Kuleshov and Precup 2000). Every planning step is regarded as an independent multi-armed bandit problem at the current state with the set of applicable actions as the arms of the bandit. The ϵ-greedy algorithm for the multi-armed bandit problem estimates the average reward of each arm by pulling the current best arm with probability $1-\epsilon$ and one of the remaining arms with probability ϵ, and finally returns the arm with the highest average reward. The ϵ-greedy algorithm of *Random-Bandit* estimates $Q^{\pi}_{h}(s,a)$ for the random policy (π) for each action (a) applicable in the current state (s) for horizon h and returns $\hat{a} = arg\max_a Q^{\pi}(s,a)$.

Introduction

The planner *Random-Bandit* has been implemented as a component of *Prost* (Keller and Eyerich 2012) as it relies on many existing functionalities in *Prost*. *Prost* is the state-of-the-art search-based online planner for *RDDL* domains. Figure 1 shows the schematic diagram of the entire planning system. *RDDLSim* (Sanner 2010) is the *RDDL* server used for evaluation in the competition. *Prost* initiates (and

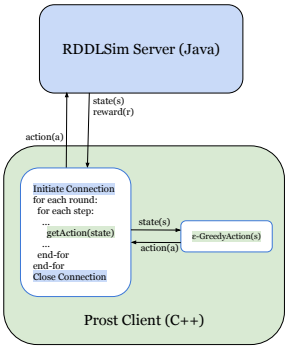

Figure 1: Schematic Diagram

also terminates) the communication with the server, receives and parses the *RDDL* domain and problem files, and initializes the required data structures. The nested *for* loops in the figure denote the evaluation loop in which *Prost* returns an action for the current state to the server and receives the reward and next state from the server. At each planning step *Prost* calls the *Random-Bandit* function ϵ-GreedyAction(s) with the current state s and returns the received action $\hat{a}$ to the server instead of invoking its own planning routines.

The ϵ-Greedy Algorithm

The ϵ-Greedy algorithm estimates $Q^{\pi}(s,a)$ for each action $a \in A_s$ applicable in state s for the random policy π for horizon h and returns $\hat{a} = arg\max_a Q^{\pi}(s,a)$. In Algorithm 1 below, the function *random-number*$(0,1)$ returns a random number between 0 and 1, *random-action*$(A_s \backslash \{\hat{a}\})$ returns a random action from the set A_s excluding action $\hat{a}$, and *next-state*(s,a) returns the next state s' and reward r as a result of taking action a in state s.

Algorithm 1 ϵ-GreedyAction(s)

Initialize $Q^{\pi}(s,a) \leftarrow 0, \forall a \in A_s$
Initialize $N(a) \leftarrow 0, \forall a \in A_s$
Initialize $\hat{a} \leftarrow$ random-action(A_s)
repeat
  $r \leftarrow$ random-number$(0,1)$
  if $r > \epsilon$ **then**
    $a \leftarrow \hat{a}$
  else
    $a \leftarrow$ random-action$(A_s \backslash \{\hat{a}\})$
  end if
  $N(a) \leftarrow N(a) + 1$
  $(s', r) \leftarrow$ next-state(s,a)
  $R \leftarrow r$
  $s \leftarrow s'$
  for $i = 1..h$ **do**
    $(s', r) \leftarrow$ next-state$(s, \pi(s))$
    $R \leftarrow R + r$
    $s \leftarrow s'$
  end for
  $Q^{\pi}(s,a) \leftarrow Q^{\pi}(s,a) + (R - Q^{\pi}(s,a))/N(a)$
  if $Q^{\pi}(s,a) > Q^{\pi}(s,\hat{a})$ **then**
    $\hat{a} \leftarrow a$
  end if
until time-limit is not reached
return $\hat{a}$

Implementation Details

The important functionalities in *Prost* used in implementing *Random-Bandit* are

1. The *IPPCClient* class for establishing (and terminating) the connection with the *RDDL* server, parsing the *RDDL* domain and problem files and initializing data structures, and running the evaluation loop receiving state and reward signals and sending actions
2. The *RandomWalk* class for simulating a trajectory from state s starting with action a and then following the random policy π for h steps accounting for steps 12 through 19 in algorithm 1
3. The *IDS* class to estimate the best rollout horizon h for the problem by means of iterative deepening search

Parameter Settings: The main parameters of the algorithm are ϵ, the rollout horizon h, and the decision-time for each planning step. ϵ is set to 0.5. The rollout horizon h is initialized to the minimum of 7 or the value returned by the *IDS* class and reduced to the number of remaining steps for planning steps near the end of an episode. The decision-time is set to 75% of the average time available for each step re-computed at the beginning of each round.

Acknowledgements

Many thanks to Dr. Thomas Keller for his help with resolving problems connected to *Prost* functionalities.

References

Keller, T., and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Kuleshov, V., and Precup, D. 2000. Algorithms for the multi-armed bandit problem. *Journal of Artificial Intelligence Research (1)* 1–48.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language description.

A2C-Plan: A Reinforcement Learning Planner

Alan Fern, Anurag Koul, Murugeswari Issakkimuthu and Prasad Tadepalli
School of EECS, Oregon State University
Corvallis, OR 97331, USA

Abstract

A2C-Plan is an offline planner that trains a policy network through Reinforcement Learning (RL) using an Advantage Actor-Critic (A2C) algorithm. It works in two phases - Training and Evaluation. In the training phase it trains a deep neural network (Goodfellow, Bengio, and Courville 2016) for the problem instance using the A2C algorithm with simulated trajectories and normalized rewards. In the evaluation phase it uses the trained network to get an action for the current state by means of a single feed-forward pass.

Introduction

Figure 1 shows the schematic diagram of the entire planning system. There are three components: *RDDLSim* (Sanner 2010), the Java *RDDL* server used for evaluation in the competition, a C++ dynamic library based on *Prost* (Keller and Eyerich 2012), and a Python component consisting of a PyTorch policy network. *Prost* is the state-of-the-art search-based online planner for *RDDL* domains. The C++ dynamic library based on *Prost* acts as an intermediate layer between the *RDDL* server and the PyTorch network providing routines for communicating with the server, running the evaluation loop and simulating trajectories during the training phase.

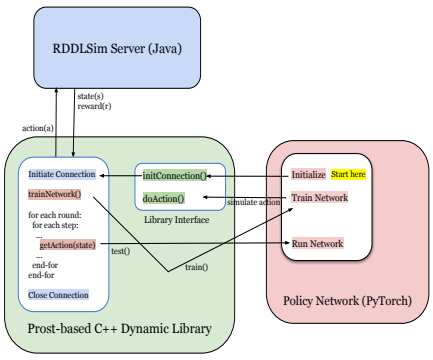

Figure 1: Schematic Diagram

The Python component starts the control flow by calling *initConnection()* in the dynamic library sending handles of callback functions *train()* and *test()* for training and running the policy network respectively. The dynamic library actually initiates (and also terminates) the communication with the server, receives and parses the *RDDL* domain and problem files, initializes the required data structures, and starts the network training process by invoking the *train()* callback function. The nested *for* loops in the dynamic library denote the evaluation loop in which it returns an action for the current state to the server and receives the reward and next state from the server. At each planning step the library invokes the *test()* callback function to run the policy network with the current state s and returns the received action $\hat{a}$ to the server.

Training the Policy Network

The policy network is a fully-connected network with two hidden layers. The input layer has as many nodes as the number of state-fluents (n) in the problem, the hidden layers have $3*n$ and $2*n$ units respectively, and the final layer has as many action nodes as the number of ground actions (m) in the problem plus an additional value node. The hidden layers have *ReLU* non-linear units and the output layer is a *softmax* layer that computes a probability distribution over the set of m actions. Figure 2 shows the network architecture for a problem with 2 state-fluents and 4 ground actions.

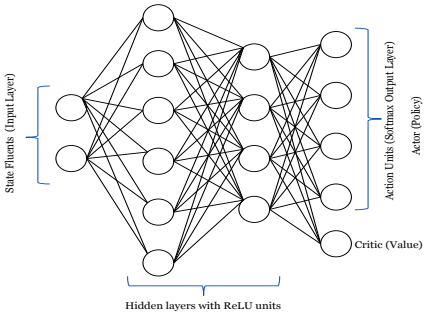

Figure 2: Network Architecture

Actor-Critic Algorithms

Actor-Critic methods (Konda and Tsitsiklis 2000) bring together the advantages of actor-only methods that directly learn a parameterized policy and critic-only methods that learn a value function by means of training a critic network using simulations and using the learned critic values to make gradient updates to the parameters of the policy network. In *A2C-Plan* the actor and critic share the same network up to the penultimate layer as shown in figure 2. The Advantage Actor-Critic (A2C) algorithm uses the *Q-Advantage* of an action $(Q(s,a) - V(s))$ instead of the value of a state $V(s)$ to update the actor parameters.

Algorithm 1 below is just meant for outlining the steps involved for one training episode. Details of the Generalized Advantage Estimation (GAE) procedure used in the implementation for updating *actor-loss* can be found in (Schulman et al. 2016). The functions *critic(s)* and *actor(s)* return the critic value and actor probabilities computed by the network respectively, *reset(env)* resets the environment and returns the initial state of an episode, *sample(P)* returns an action sampled using the probability distribution P computed by the actor network along with the probability p_a of the selected action a, and *max* and *min* are the maximum and minimum reward values for the problem computed or approximated by *Prost*. The entropy term e in the actor loss function encourages exploration.

Algorithm 1 A2C(*net*, *env*)

repeat
  Create Arrays V, R, L, E
  Initialize $i \leftarrow 0$, $s \leftarrow$ reset(*env*)
  while not end-of-episode **do**
    $v \leftarrow$ *critic*(s), $P \leftarrow$ *actor*(s)
    $(a, p_a) \leftarrow$ sample(P)
    $l \leftarrow log(p_a)$
    $e \leftarrow \sum_{p_a \in P} log(p_a)$
    $(s', r) \leftarrow$ next-state(s, a)
    $r \leftarrow r/(max - min)$
    $V[i] \leftarrow v$, $R[i] \leftarrow r$
    $L[i] \leftarrow l$, $E[i] \leftarrow e$
    $s \leftarrow s'$, $i \leftarrow i + 1$
  end while
  critic-loss $\leftarrow$ *actor-loss* $\leftarrow 0$
  $\hat{v} \leftarrow 0$
  for $i = H \ldots 1$ **do**
    $\hat{v} \leftarrow \gamma\hat{v} + R[i]$
    critic-loss $\leftarrow$ *critic-loss* + $(V[i] - \hat{v})^2$
    Adv $\leftarrow R[i] + \gamma V[i+1].value - V[i].value$
    actor-loss $\leftarrow$ *actor-loss* - *Adv* * $L[i] + E[i]$
  end for
  Minimize *critic-loss*, *actor-loss*
until time-limit or memory-limit is not reached

Implementation Details

The Python - C++ interface is implemented using the *ctypes* library (https://docs.python.org/3/library/ctypes.html). The important functionalities in *Prost* used in the dynamic library are

1. The *IPPCClient* class for establishing (and terminating) the connection with the *RDDL* server, parsing the *RDDL* domain and problem files and initializing data structures, and running the evaluation loop receiving state and reward signals and sending actions
2. The *SearchEngine* class functions estimating the maximum and minimum rewards for the problem instance
3. All the classes involved in simulating an action at a given state to compute the reward and the next state

Parameter Settings:

1. The competition imposes a *RAM* limit of 4GB for the process. *RAM* usage is periodically monitored in the Python program while training the network and the training process is terminated once a limit of 3.5GB is reached.
2. The total time (T) available to solve a problem instance needs to be divided between the training and evaluation phases leaving enough time for other associated computations like the initial parsing process. Approximately 70% of the total time T is set aside for just training the network. To be precise, an untrained network is run for one-fifth (15 for the competition) of the total number of episodes (75 in the competition) to compute a time t and time for final evaluation (t_e) is set to $2 \times t$ times the total number of episodes and time for training is set to 75% of $T - T_e$.
3. For domains with action pre-conditions some of the actions might not be applicable at a given state. When that happens during training or evaluation an applicable action with the highest probability is used instead.

Acknowledgements

Many thanks to Dr. Thomas Keller for his help with resolving problems connected to *Prost* functionalities.

References

[Goodfellow, Bengio, and Courville 2016] Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep learning*. MIT press.

[Keller and Eyerich 2012] Keller, T., and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

[Konda and Tsitsiklis 2000] Konda, V., and Tsitsiklis, J. 2000. Actor-critic algorithms. In *SIAM Journal on Control and Optimization*, 1008–1014. MIT Press.

[Sanner 2010] Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language description.

[Schulman et al. 2016] Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; and Abbeel, P. 2016. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

PROST-DD - Utilizing Symbolic Classical Planning in THTS

Florian Geißer and **David Speck**
University of Freiburg, Germany
{geisserf, speckd}@informatik.uni-freiburg.de

Abstract

We describe PROST-DD, our submission to the International Probabilistic Planning Competition 2018. Like its predecessor PROST, which already participated with success at the previous IPPC, PROST-DD is based on the trial-based heuristic tree search framework and applies the $UCT^\star$ algorithm. The novelty of our submission is the heuristic used to initialize newly encountered decision nodes. We apply an iterative symbolic backward planning approach based on the determinized task. Similarly to the SPUDD approach and recent work in symbolic planning with state-dependent action costs, we encode costs and reachability of states in a single decision diagram. During initialization, these diagrams are then used to query a state for its estimated expected reward. One benefit of this heuristic is that we can optionally interweave the standard heuristic of PROST, the IDS heuristic.

Introduction

The 6th edition of the International Probabilistic Planning competition initially consisted of three different tracks: the discrete MDP track, the continous MDP track, and the discrete SSP track. In this paper, we will discuss our submission to the discrete MDP track, which consists of a novel heuristic implemented into the PROST planner (Keller and Eyerich 2012), the winner of the previous IPPC. The goal of the discrete SSP track is to come up with a policy for a factored Markov decision process (MDP) with fixed initial state and fixed horizon. The reward is state-dependent and there are no dead-ends. As in the previous IPPC, the language to model the planning tasks is the Relational Dynamic Influenc Diagram (RDDL) language (Sanner 2010), and planners are evaluated by executing 75 runs per instance and comparing the average accumulated reward.

The PROST planner is based on the trial-based heuristic tree search framework (THTS) (Keller and Helmert 2013) which allows to mix several ingredients to compose an anytime optimal algorithm for finite-horizon MDPs. One of these ingredients is the state-value initialization (or: heuristic) used to give an initial estimate for previously unknown states. Our submission exchanges the iterative deepening search (IDS) heuristic, the original heuristic implemented in PROST, with a heuristic based on backward symbolic search (BSS) on the determinized task. On the one hand, this approach can be compared to SPUDD (Hoey et al. 1999), a stochastic planning approach using decision diagrams. On the other hand, it can be compared to recent work on symbolic planning for tasks with state-dependent action costs (Speck, Geißer, and Mattmüller 2018).

Before we describe our heuristic, we quickly introduce the THTS framework and the setup of the PROST planner in the previous IPPC. The next section then explains the BSS heuristic, before we finally sketch how we can interweave BSS and IDS, to come up with a stronger heuristic for more challenging tasks.

Trial-based Heuristic Tree Search

The trial-based heuristic tree search (THTS) framework (Keller and Helmert 2013) allows to model several well-known probabilistic search algorithms in one common framework. It is based on the following ingredients: *heuristic function*, *backup function*, *action selection*, *outcome selection*, *trial length*, and *recommendation function*. Independent of the specific ingredients, the general tree search algorithm maintains a tree of alternating decision and chance nodes, where a decision node contains a state s and a state-value estimate based on previous trials. A chance node contains a state s, an action a and a Q-value estimate, which estimates the expected value of action a applied in state s. The algorithm performs so-called trials, until it either computed the optimal state-value estimate of the state in the root node of the tree, or until it is out of time. A THTS trial consists of different phases: the *selection phase* traverses the tree according to action and outcome selection until a previously unvisited decision node is encountered. Then, in the *expansion phase*, this selected node is expanded, where for each action a child node is added to the tree and initialized with a heuristic value according to the heuristic function. The trial length parameter decides if the selection phase starts again, or if the *backup phase* is initiated. In this phase, the visited nodes are updated in reversed order according to the backup function. A trial finishes when the backup function is called on the root node. In the case that the algorithm is out of time, the *recommendation function* recommends which action to take, based on the values of the child nodes. For more information on the THTS algorithm we recommend the original THTS paper (Keller and Helmert 2013), as well as the PhD thesis of T. Keller (Keller 2015) which introduces recommendation functions and contains a thorough theoretical

and empirical evaluation of a multitude of algorithms realized within this framework.

Our submission is based on the PROST configuration of the IPPC 2014, together with a novel heuristic function. Before we describe this heuristic, we quickly mention the other ingredients used in our configuration. The action selection function is based on the well-known UCB1 formula (Auer, Cesa-Bianchi, and Fischer 2002) which has a focus on balancing exploration versus exploitation. The outcome selection is based on Monte-Carlo sampling and samples outcomes according to their probability, with the additional requirement that the outcome was not already marked as solved by the backup function. This backup function is a combination of Monte-Carlo backups and Full Bellman backups, and weights outcomes proportionally to their probability. It allows for missing (i.e. non-explicated) outcomes and also for labeling nodes as solved, where a node is solved if its optimal value estimation is known. These ingredients are the same used in the IPPC 2014. For the recommendation function, we apply the *most played arm* recommendation (Bubeck, Munos, and Stoltz 2009), which recommends one of the actions that have been selected most often in the root node (uniformly at random). This recommendation function was shown (Keller 2015) to be superior in combination with the other ingredients.

Backward Symbolic Search Heuristic (BSS)

The *Backward Symbolic Search Heuristic (BSS)* exploits the efficiency of symbolic search and the compactness of symbolic data structures in form of decision diagrams. More precisely, we use Algebraic Decision Diagrams as the underlying symbolic data structure. *Algebraic Decision Diagrams* (Bahar et al. 1997) represent algebraic functions of the form $f : \mathcal{S} \to \mathbb{R} \cup \{-\infty, \infty\}$. Formally, an ADD is a directed acyclic graph with a single root node and multiple terminal nodes. Internal nodes correspond to binary variables, and each node has two successors. The *low edge* represents that the current variable is false, while the *high edge* represents that the current variable is true. Evaluation of a function then corresponds to the traversal of the ADD according to the assignment of the variables.

The main idea of BSS is to determinize a given MDP while representing the MDP as decision diagrams with a subsequent backward exploration of the state space. Recently, Speck, Geißer, and Mattmüller (2018) showed how symbolic search can be applied to deterministic planning tasks with state-dependent action costs. Similar to their work, we perform a symbolic backward search on the determinized MDP which corresponds to a classical planning task with state-dependent action costs. This backward search results in multiple ADDs, where each ADD represents states associated with the maximum reward that can be achieved from the corresponding states. More precisely, BSS can be divided into three parts. First, a given MDP is determinized (all-outcome or most-likely). Second, a backward breadth-first search is performed, with the number of backward planning steps equal to the horizon. In each backward planning step we obtain an ADD which maps reachable states to rewards (*symbolic layers*). Finally, during the actual search the precomputed rewards (stored in decision diagrams) are used to evaluate state actions pairs, i.e. Q-values. In the following, we will explain each step in more detail.

Let a be an action of a given MDP. Action a has an empty precondition and effects $p(x' := \neg x) = 1$, $p(y' := 1) = 0.6$ and $p(y' := 0) = 0.4$. In other words, action a always negates the value of x and sets y to 1 (0) with probability of 0.6 (0.4). Finally, the reward function of action a is defined as $R(s, a) = 2 + 5 \cdot s(y)$, where $s(y)$ is the value of y in state s. In the initial step, action a is determinized (here: most-likely determinization) and represented as a transition relation in form of an ADD mapping state pairs consisting of predecessors S and successors S' to 1 (true) or 0 (false). Figure 1 depicts the ADD which represents action a as transition relation after applying the most-likely determinization, i.e. only outcomes with a probability of 0.5 are considered [1]. Finally, the reward function is added to the transition relation of a. If we transform the reward to a negative value, we obtain a transition relation representing costs which is analogous to the formalization of Speck, Geißer, and Mattmüller (2018). This transformation of an action is applied to each action which results in a determinized planning task with state-dependent action costs.

The symbolic backward search starts with all states associated with zero costs as shown on the left side of Figure 2. We perform h backward steps where h is equal to the horizon. Each backward step creates a symbolic Layer L_i, which stores for each state the maximal reward which can be achieved in the remaining i steps. In Figure 2, after one backward step (L_1) we can obtain a reward of 2 or 7 by applying an action. Note that there can be states where no action can be applied which is represented by a reward of $-\infty$. Finally, we initialize the value of a state s with action a as follows: let i be the number of remaining steps and let $S'_{s,a}$ be the set of possible successor states of applying action a in state s, i.e. $S'_{s,a} = \{s' | p(s'|s, a) > 0\}$. The initial Q-value of a state action pair is defined as

$$Q^{\textbf{init}}(s, a) = \frac{\sum_{s' \in S'_{s,a}} L_{i-1}(s')}{|S'_{s,a}|}.$$

In other words, we take the average of the precomputed rewards of all successor states of predecessor state s with respect to action a. In the following, we present how we can combine this heuristic with the usual forward search heuristic applied by the PROST planner.

Combining explicit forward and symbolic backward search heuristics

One property of the symbolic backward search heuristic is that it explores the whole deterministic task and collects all states and rewards reachable from the end of the horizon. While this is certainly easier for the determinized task it still is a hard problem and as a result we might only have results for parts of the horizon. In this case, we can make use of

[1] This may contrast with some other notions of most-likely, namely that most-likely usually means only accepting the most-likely outcome.

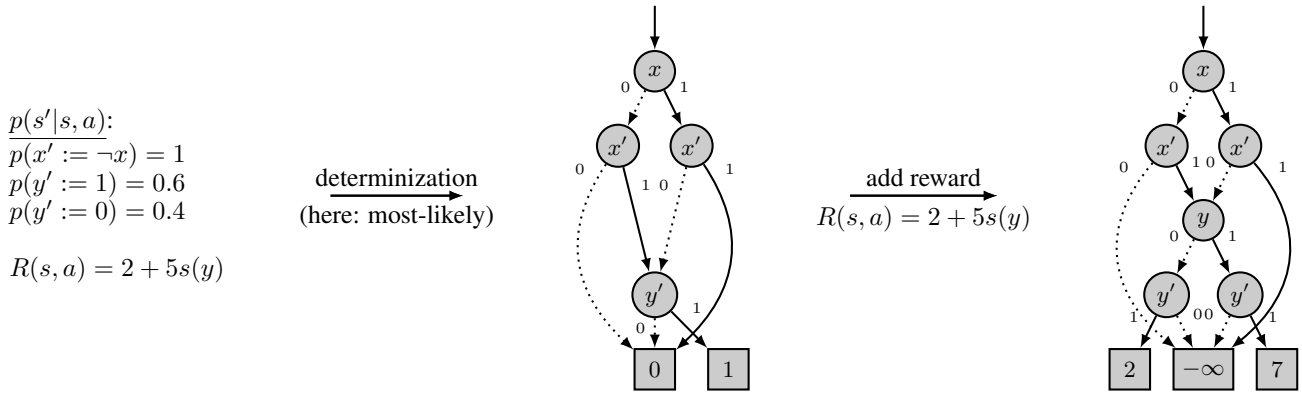

Figure 1: Transformations of action a. Action a has an empty precondition and effects $p(x' := \neg x) = 1$, $p(y' := 1) = 0.6$ and $p(y' := 0) = 0.4$. Functions related to action a are depicted as ADDs. In the middle the determinized transition relation and on the right the final transition relation with rewards.

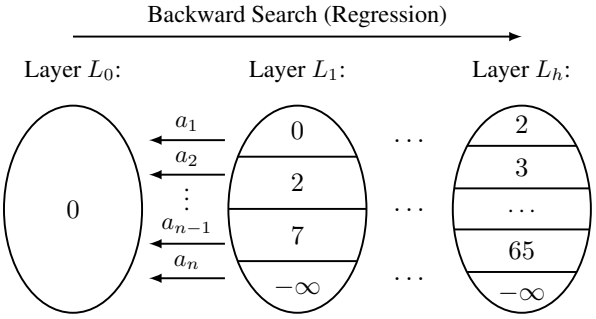

Figure 2: Visualization of symbolic backward search starting with all states associated with zero costs. At each backward step, all states leading to the previous state s are stored in an ADD called Layer L_i and mapped to the maximum reward which can be achieved from s in the remaining i steps.

the original heuristic implemented in the PROST planner, which is based on iterative deepening search (IDS).

The IDS heuristic also performs a determinization of the probabilistic task, and then conducts a depth-first search to compute the maximal reward reachable in the *next* d steps. The value of d is computed before search starts and usually depends on the complexity of the domain and its transition functions. The IDS heuristic can therefore be seen as an explicit forward search algorithm which complements our symbolic backward search approach. While our heuristic computes maximal rewards from the end of the horizon up to some step i, IDS computes the maximal reward reachable in the next d steps. Therefore, whenever we are not able to compute all layers, we combine both heuristics by querying the backward search value of the last layer and add the value estimated by IDS (with depth corresponding to the maximal layer). This can also be seen as a portfolio approach: if we are able to build all layers in the symbolic search we rely solely on our heuristic. If the task is so complex that we are not even able to build a single layer we only rely on the IDS heuristic (and have a setup very similar to the previous IPPC configuration). In all other cases we interweave both heuristics in order to generate an initial state value estimate which is better than when we would solely rely on a single heuristic.

Competition Analysis

Now that the competition is over we present a brief analysis of some of the results (Keller 2018b). The two versions of our planner differ only in the heuristics. Version 1 computes the BSS heuristic based on a most-likely determinization, while version 2 is based on an all-outcome determinization. We are especially interested in a comparison to the baseline planners (the PROST planner configurations of 2011 and 2014), since our planner mostly differs in the heuristic.[2] This year's IPPC consisted of eight domains with 20 instances each, resulting in a total of 160 instances. It turns out that grounding was a major challenge this year. For example, the hardest Academic Advising instance has more than 11 *billion* grounded actions, due to the combinatorial blowup (5 out of 269 actions are applicable concurrently). In total, our planner was unable to ground 31 instances. This certainly warrants investing more research effort into the grounding of concurrent actions. Regarding the remaining instances, the PROST-DD planner crashed during search in 16 instances, mainly in the Earth observation domain due to a bug. Table 1 shows the average rewards of our planner (bug fixed) compared to the PROST versions of the IPPCs 2011 and 2014 on the Earth Observation domain. The differences in performance are minor.

[2]We additionally fixed some bugs of the PROST planner which were mostly concerned with not exceeding the memory limit (the baseline planners only used 2GB RAM). Unfortunately, we introduced a bug which led to a crash in most of the Earth Observation domain instances; otherwise our planner's score would have exceeded the baseline score.

	PROST-DD		PROST	
ID	most-likely (v1)	all-outcome (v2)	2011	2014
1	-8.84	-8.92	-15.95	**-8.49**
2	-486.75	-483.91	**-478.11**	-484.93
3	-704.89	-709.97	**-697.33**	-714.44
4	**-1574.65**	-1600.60	-1591.32	-1616.31
5	-649.13	**-644.63**	**-643.91**	-672.15
6	-239.01	**-236.27**	-237.88	-248.41
7	**-39.40**	**-39.65**	-40.79	-42.04
8	-431.67	**-429.09**	-455.09	-455.47
9	-1355.19	-1355.19	-1291.91	**-1278.41**
10	-3203.95	-3222.25	**-3167.59**	**-3163.08**
11	**-820.28**	**-819.43**	-833.20	-825.21
12	-1668.00	-1693.24	**-1657.35**	-1665.31
13	-1919.39	-1922.29	**-1841.96**	**-1839.17**
14	-10099.60	-10103.80	-9957.04	**-9827.79**
15	-2645.88	-2627.17	**-2588.33**	-2775.01
16	-353.41	**-351.64**	-380.27	-368.68
17	-1875.48	-1875.48	-1791.63	**-1736.52**
18	-3186.06	-3186.60	-2989.44	**-2843.33**
19	-5170.25	-5114.09	-4954.21	**-4825.60**
20	-12702.90	-12665.50	-12731.80	**-12622.90**

Table 1: Average reward of the PROST-DD planner (version 1 and version 2) compared to the PROST versions of the IPPCs 2011 and 2014 on the Earth Observation domain.

The key question remains: has the BSS heuristic paid off? To answer this question, we analyze the number of tasks for which it was possible to compute at least one layer, which meant that the BSS heuristic could also be used during the search. Unfortunately, it turns out that it was only possible to compute at least one layer in 23 instances. This is certainly due to the fact that the domains of this years IPPC were more challenging compared to previous problems of former IPPCs. The BSS heuristic was mostly succesful in the domains Academic Advising and Push Your Luck. In both domains, the performance of both configurations was evenly good, and superior to other planners. The heuristic computation took around 10% of the search time. In the Manufacturer and Cooperative Recon domains the heuristic was unable to generate a single layer and thus consumed time in the precomputation phase without providing useful information. This might be a reason for the low performance. However, once computed the BSS heuristic is informative and helpful. This certainly shows that the heuristic has potential, but needs to be more efficient, especially when faced with large and difficult problems. We already have some ideas for such improvements. Interestingly, we also outperformed other planners in Wildlife Preserve, even though we use the same heuristic as the baseline planner in this case. This may be due to the additional memory we use, but also due to some modifications to the grounding of actions which differs slightly from the baseline.

In summary, the heuristic presented here has paid off in some domains and has affected the planner's performance in others due to loss of time. PROST-DD proved to be a competitive planner and the BSS heuristic showed promising results.

Our planner submission is available in the official IPPC repositories on Bitbucket (Keller 2018a). We fixed the bug which led to crashes in the Earth Observation domain in the branch ipc2018-disc-mdp. The original competition version is available on the branch ipc2018-disc-mdp-competition.

Acknowledgments

David Speck was supported by the German National Science Foundation (DFG) research unit FOR 1513 on Hybrid Reasoning for Intelligent Systems (`http://www.hybrid-reasoning.org`).

References

[Auer, Cesa-Bianchi, and Fischer 2002] Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finitetime Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47:235–256.

[Bahar et al. 1997] Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1997. Algebraic decision diagrams and their applications. In *Proceedings of the International Conference on Computer Aided Design (ICCAD 1993)*, volume 10, 171–206.

[Bubeck, Munos, and Stoltz 2009] Bubeck, S.; Munos, R.; and Stoltz, G. 2009. Pure Exploration in Multiarmed Bandits Problems. In *Algorithmic Learning Theory, 20th International Conference (ALT 2009)*, 23–37.

[Hoey et al. 1999] Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, 279–288.

[Keller and Eyerich 2012] Keller, T., and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*, 119–127.

[Keller and Helmert 2013] Keller, T., and Helmert, M. 2013. Trial-based Heuristic Tree Search for Finite Horizon MDPs. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 135–143.

[Keller 2015] Keller, T. 2015. *Anytime Optimal MDP Planning with Trial-based Heuristic Tree Search*. Ph.D. Dissertation, University of Freiburg.

[Keller 2018a] Keller, T. 2018a. Bitbucket repository of the ippc 2018 planners. `https://bitbucket.org/account/user/ipc2018-probabilistic/projects/EN`. [Online; accessed 08-October-2018].

[Keller 2018b] Keller, T. 2018b. Presentation slides of the ippc 2018. `https://ipc2018-probabilistic.bitbucket.io/results/presentation.pdf`. [Online; accessed 08-October-2018].

[Sanner 2010] Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description.

[Speck, Geißer, and Mattmüller 2018] Speck, D.; Geißer, F.; and Mattmüller, R. 2018. Symbolic Planning with Edge-Valued Multi-Valued Decision Diagrams. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*, 250–258.