Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)

Self-Taught Decision Theoretic Planning with First Order Decision Diagrams

Saket Joshi
Dept. of Computer Science
Tufts University
sjoshi01@cs.tufts.edu

Kristian Kersting
Knowledge Discovery Department
Fraunhofer IAIS, Sankt Augustin, Germany
kristian.kersting@iais.fraunhofer.de

Roni Khardon
Dept. of Computer Science
Tufts University
roni@cs.tufts.edu

Abstract

We present a new paradigm for planning by learning, where the planner is given a model of the world *and* a small set of states of interest, but no indication of optimal actions in these states. The additional information can help focus the planner on regions of the state space that are of interest and lead to improved performance. We demonstrate this idea by introducing novel model-checking reduction operations for First Order Decision Diagrams (FODD), a representation that has been used to implement decision-theoretic planning with Relational Markov Decision Processes (RMDP). Intuitively, these reductions modify the construction of the value function by removing any complex specifications that are irrelevant to the set of training examples, thereby focusing on the region of interest. We show that such training examples can be constructed on the fly from a description of the planning problem thus we can bootstrap to get a self-taught planning system. Additionally, we provide a new heuristic to embed universal and conjunctive goals within the framework of RMDP planners, expanding the scope and applicability of such systems. We show that these ideas lead to significant improvements in performance in terms of both speed and coverage of the planner, yielding state of the art planning performance on problems from the International Planning Competition.

Introduction

Planning under uncertainty is an important problem of Artificial Intelligence and has many relevant applications in medicine, robotics, game-playing, scheduling and logistics among others. Simple straight line plans from the start state to the goal cannot guarantee achievement of the goal when the underlying world is probabilistic. Therefore, it is not surprising that there have been a number of approaches and systems that address this problem. Recently the advent of the International Planning Competition (IPC) has encouraged the development of efficient systems that plan in stochastic domains. Although the most successful of these systems are based on forward heuristic search from the starting state (Yoon, Fern, and Givan 2007; Teichteil-Koenigsbuch, Infantes, and Kuter 2008), an alternative approach of generating state and action utilities by reasoning backwards from the goal state has recently received attention. Typically systems following this method (Kersting, Otterlo, and De Raedt 2004; Hölldobler, Karabaev, and Skvortsova 2006; Sanner and Boutilier 2009; Joshi and Khardon 2008) are based on Boutilier, Reiter, and Price's (2001) Symbolic Dynamic Programming (SDP) algorithm and they solve a problem that is more general than the one at hand. These systems represent the planning domain as a relational Markov decision process (RMDP) and solve it by backward reasoning to generate a lifted value function over structures in the domain. Such a value function can then be applied to solve any planning problem in the domain irrespective of the actual number of domain objects in the problem. Therefore, although solving RMDPs is expensive, the cost of generating the value function can be amortized over a large number of planning problems.

One characteristic of systems based on the SDP algorithm is the need for logical simplification in the process of backward reasoning. Naive backward reasoning introduces redundancies in the structure of the value function and these structural redundancies have to be removed in order to maintain a value function of reasonable size. To date all such systems have used theorem proving to perform the simplification or reduction of the intermediate representation. This process is expensive and in addition requires extra knowledge about the domain (e.g., saying that a box cannot be in two cities at the same time) to be encoded. This makes for slow and less general systems. In this paper we show that both of these issues can be mitigated by changing the focus to simplification based on model-checking.

Specifically, we introduce a new paradigm for planning by learning: the planner is given a model of the world *and* a small set of states of interest, but *no* indication of optimal actions in any states. Similar to Price and Boutilier's (2003) implicit imitation for reinforcement learning, the additional information in the states can help focus the planner on regions of the state space that are of interest and lead to improved performance. We give a concrete instantiation of this idea in the context of the FODD-PLANNER (Joshi and Khardon 2008), which is based on First Order Decision Diagrams (FODD) (Wang, Joshi, and Khardon 2008). One of the main contributions of this paper is to introduce two

new heuristic reductions that simplify FODDs, by removing edges and nodes, in the context of a given set of examples. These reductions can be seen as a practical equivalent of the sound and complete but expensive model-checking reduction introduced in our previous work (Joshi, Kersting, and Khardon 2009). Our new reductions give a significant speedup and makes the SDP approach practically applicable to a wide range of problems.

Focused planning as outlined above requires a set of training examples. As a second contribution we show that such training examples can be constructed on the fly from a description of the planning problem. Thus we can bootstrap our planner to get a self-taught planning system. We propose several such approaches, based on backward random walks from goal states enhanced with specific restrictions to ensure coverage of a rich set of states with a small sample. There have been other approaches where training examples have been used to generate models for solving planning problems (Fern, Yoon, and Givan 2003; Gretton and Thiebaux 2004). However, our approach differs from these in that our training set does not need information about optimal actions or values along with the examples.

The final contribution of the paper provides a new heuristic to embed universal and conjunctive goals within the framework of RMDP planners. Previous work on RMDP planners (Sanner and Boutilier 2009; Joshi and Khardon 2008) simplifies the offline planning stage by planning for each goal atom separately. The result is combined by adding the values from each goal atom. However, this is limited and does not apply well to many domains. To overcome this, we propose a new weighting heuristic based on goal ordering. Our method uses static analysis to calculate a weighted composition of individual goal values. As we show in the experiments this leads to significant improvements in performance in domains with interacting dependent sub-goals.

We provide an extensive evaluation of these ideas in the context of the FODD-PLANNER system. The experiments characterize performance in terms of different system ingredients, compare the FODD-PLANNER (Joshi and Khardon 2008) to the self-taught model-checking variants, and evaluate the system on challenge problems from IPC. Results show state of the art performance on these problems and a significant improvement in efficiency over the original FODD-PLANNER.

Preliminaries

Relational Markov Decision Processes

A Markov decision process (MDP) is a mathematical model of the interaction between an agent and its environment (Puterman 1994). Formally a MDP is a 4-tuple $<S, A, T, R>$ defining a set of states S, set of actions A, a transition function T defining the probability $P(s' \mid s, a)$ of getting to state s' from state s on taking action a, and an immediate reward function $R(s)$. The objective of solving a MDP is to generate a policy that maximizes the agent's total, expected, discounted, reward. Intuitively, the expected utility or value of a state is equal to the reward obtained in the state plus the discounted value of the state reached by the best action in the state. This is captured by the Bellman equation as $V(s) = Max_a[R(s) + \gamma\Sigma_{s'}P(s' \mid s, a)V(s')]$. The Value Iteration (VI) algorithm is a dynamic programming algorithm that treats the Bellman equation as an update rule and iteratively updates the value of every state until convergence. Once the optimal value function is known, a policy can be generated by assigning to each state the action that maximizes expected value.

A relational MDP is a MDP where the state space is described by a set of objects and relations among them. Therefore a state is an interpretation in the terminology of logic programming and a model or structure in first order logic. The actions and transitions are typically described by some schema parametrized by objects. An action schema and a set of concrete objects induces a concrete action in the domain. Similarly the transition function is described by parametrized schema. The reward function (goal in planning context) is described by a numerical function over states, often using logical formulas to describe partitions of the state space with specific values. Stochastic planning in structured domains is a prime example of RMDPs and fits precisely in this description. In this case we can obtain an abstract solution for all potential instances for the domain. This is the type of solution provided by dynamic programming algorithms for RMDPs.

For propositional problems Hoey et al. (1999) showed that if $R(s)$, $P(s' \mid s, a)$ and $V(s)$ can be represented using algebraic decision diagrams (ADDs), then VI can be performed entirely using the ADD representation thereby avoiding the need to enumerate the state space. Later Boutilier, Reiter, and Price (2001) developed the Symbolic Dynamic Programming (SDP) algorithm in the context of situation calculus. This algorithm provided a framework for dynamic programming solutions to relational MDPs that was later employed in several formalisms and systems (Kersting, Otterlo, and De Raedt 2004; Hölldobler, Karabaev, and Skvortsova 2006; Sanner and Boutilier 2009; Wang, Joshi, and Khardon 2008). One of the important ideas in SDP was to represent stochastic actions as deterministic alternatives under nature's control. This helps separate regression over deterministic action alternatives (step 1 below) from the probabilities of action effects (step 2). This segregation is necessary in order to simplify computation when transition functions are represented as relational schema. The relational VI algorithm is as follows:

1. **Regression:** Regress the n step-to-go value function V_n over every deterministic variant $A_j(\vec{x})$ of every action $A(\vec{x})$ to produce $Regr(V_n, A(\vec{x}))$.
2. **Add Action Variants:** Generate the Q-function $Q_{V_n}^{A(\vec{x})} = R \oplus [\gamma \otimes \oplus_j (prob(A_j(\vec{x})) \otimes Regr(V_n, A_j(\vec{x})))]$ for each action $A(\vec{x})$.
3. **Object Maximization:** Maximize over the action parameters of $Q_{V_n}^{A(\vec{x})}$ to produce $Q_{V_n}^{A}$ for each action $A(\vec{x})$, thus obtaining the value achievable by the best ground instantiation of $A(\vec{x})$.
4. **Maximize over Actions:** Generate the $n+1$ step-to-go value function $V_{n+1} = \max_A Q_{V_n}^{A}$.

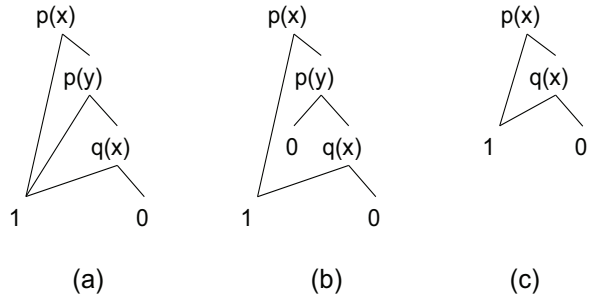

Figure 1: Examples of FODDs and edge and node removal reductions. Left going edges represent the `true` branches and right edges are the `false` branches

Wang, Joshi, and Khardon (2008) represent all the intermediate constructs in this algorithm (R, V, Q, Pr) by FODDs and all operations over the functions represented by them are performed by FODD operations. This leads to a compact implementation of the SDP algorithm.

First Order Decision Diagrams (FODD)

This section gives an overview of FODDs (Wang, Joshi, and Khardon 2008). We use standard terminology from first order logic (Lloyd 1987). A First order decision diagram is a labeled directed acyclic graph, where each non-leaf node has exactly 2 outgoing edges with `true` and `false` labels. The non-leaf nodes are labeled by atoms generated from a predetermined signature of predicates, constants and an enumerable set of variables. Leaf nodes have non-negative numeric values. The signature also defines a total order on atoms, and the FODD is ordered with every parent smaller than the child according to that order. Examples of FODDs are given in Figure 1.

Thus, a FODD is similar to a formula in first order logic. Its meaning is similarly defined relative to interpretations of the symbols. An *interpretation* defines a domain of objects, identifies each constant with an object, and specifies a truth value of each predicate over these objects. In the context of relational MDPs, an interpretation represents a state of the world with the objects and relations among them. Given a FODD and an interpretation, a *valuation* assigns each variable in the FODD to an object in the interpretation. Following Groote and Tveretina (2003), the semantics of FODDs are defined as follows. If B is a FODD and I is an interpretation, a valuation ζ that assigns a domain element of I to each variable in B fixes the truth value of every node atom in B under I. The FODD B can then be traversed in order to reach a leaf. The value of the leaf is denoted $Map_B(I, \zeta)$. $Map_B(I)$ is then defined as $max_\zeta Map_B(I)$, i.e. an aggregation of $Map_B(I, \zeta)$ over all valuations ζ. For example, consider the FODD in Figure 1(a) and the interpretation I with objects a, b and where the only true atoms are $p(a), q(a)$. The valuations $\{x/a, y/a\}$, $\{x/a, y/b\}$, $\{x/b, y/a\}$, and $\{x/b, y/b\}$, will produce the values 1, 1, 1, and 0 respectively. By the max aggregation semantics, $Map_B(I) = max\{1, 1, 1, 0\} = 1$. Thus, this FODD is equivalent to the formula $\exists x, \exists y, p(x) \vee p(y) \vee q(x)$. In general, max aggregation yields existential quantification when leaves are binary. When using numerical values we can similarly capture value functions for relational MDPs.

A descending path ordering (DPO) for FODD B is an ordered list of all paths from the root to a leaf in B, sorted in descending order by the value of the leaf reached by the path. The relative order of paths reaching the same leaf is unimportant as long as it is fixed (Joshi and Khardon 2008)

Akin to ADDs, FODDs can be combined under arithmetic operations, and reduced in order to remove redundancies. Groote and Tveretina (2003) introduced four reduction operators and these were later augmented with eight more (Wang, Joshi, and Khardon 2008; Joshi and Khardon 2008; Joshi, Kersting, and Khardon 2009). Reductions may be classified into 2 categories - those that focus on the removal of edges and those that focus on the removal of nodes. Figure 1 shows an example of edge and node removal reductions. The path $\neg p(x), p(y) \rightarrow 1$ in Figure 1(a) is redundant because if there is a valuation traversing it, there is always another valuation traversing the path $p(x) \rightarrow 1$ and achieving the same value. An applicable edge removal reduction notices this fact and replaces the 1 leaf with a 0 leaf producing the FODD in Figure 1(b). The edge removal reduction does not, however, remove the redundant node $p(y)$ because it lies on another path that is not redundant. An applicable node removal reduction notices that the removal of the node $p(y)$ does not affect the map of any interpretation and removes it to produce the FODD in Figure 1(c).

Model-Checking Reductions

Recently we introduced the R12 algorithm for reducing FODDs by model-checking (Joshi, Kersting, and Khardon 2009). The main idea is to follow the semantics when performing the reduction. If we can evaluate the FODD on all possible interpretations then portions of the diagram that are not *instrumental* in determining the final value on any of these can safely be removed. The R12 reduction embeds this idea without enumerating all possible interpretations. They show that by careful accounting one can store just enough information to yield a sound and complete reduction that replaces as many edges as possible with a zero leaf. Unfortunately, while finite, the complexity of this reduction is too high. For a FODD with n variables, it requires enumeration of n^n valuations to hypothetical interpretations. In the following we present two simple heuristic variants of this idea that allow us to remove edges as well as nodes in FODDs.

In this paper we assume a given set of "focus states" that together capture all important variation in the state space and only evaluate the diagram on these states. Note that we do not assume all states of interest are given. Just that if an important condition exists for the domain then it is realized in at least one of the given states. This is a much weaker condition. The result is an efficient variant of R12 and an extension of R12 for node removal. Our reductions reduce at least at least as much as the original R12. In general we lose soundness, i.e. we over-prune, but using the intuitive argument above, one can formalize conditions under which the reduction is also sound.

Edge Removal by Model-Checking

For edge removal we want to determine when an edge pointing to a sub-diagram can be replaced with a zero leaf. Recall that $Map_B(I) = max_\zeta Map_B(I)$. Therefore the values provided by the non maximizing ζ's can be reduced without changing the final results. We define an edge to be instrumental if it participates in a path that gives the final value on some interpretation. As explained above this is approximated by being instrumental on the given examples. In the following, conjunction p subsumes example e if there is a substitution θ such that $p\theta \subseteq e$. Edge removal can be easily implemented as follows.

Procedure 1 *R12-edge*
Input: FODD B, Sample E
Output: Reduced FODD B'

1. *Generate a DPO P for B.*
2. *$I = \{\}$*
3. *For each example e in E,*
   For $i = 1$ to $|P|$,
   if p_i subsumes e, then $I = I \cup$ edges(p_i); break
4. *For each edge e' in B such that $e' \notin I$, set target(e') $= 0$*

Clearly every path identified as instrumental and added to I is instrumental. Therefore we prune all unnecessary edges. On the other hand if the example set is poor, we may over-prune the FODD. Notice that, as long as the given examples satisfy domain constraints, we will automatically prune any paths violating such constraints (i.e. illegal abstract states) without the need to employ complex background knowledge. Therefore unlike theorem proving reductions, we do not have to maintain background knowledge and handle the complications it entails. This is an important property of model-checking reductions.

Node Removal by Model-Checking

While edge removal is important it does not handle a common type of redundancy that arises often due to the structure of the SDP algorithm where we add or multiply functions with similar structure that are standardized apart (steps 2 and 4 of the SDP algorithm). Often we have an irrelevant node above an important portion of the diagram. We cannot remove the edge from that node because it will cut off the important sub-diagram. Instead what we need is a reduction that can skip the irrelevant node (e.g., Figure 1(b) to (c)). We use this idea for nodes where one child is zero and the other is a diagram. The question is whether connecting the node's parents directly to the non-zero child will change $Map_B(I) = max_\zeta Map_B(I)$. The only way this can happen is if a valuation ζ that previously went to the zero child is now directed to a non-zero leaf which is greater than the previous maximum. As above, this condition is easy to check directly on the given set of examples.

Procedure 2 *R12-node*
Input: FODD B, Sample E, Set of Candidate nodes C
Output: Reduced FODD B'

1. *Generate a DPO P for B.*
2. *For each node $n \in C$ do the following:*
   (a) Remove node n from B by connecting the parents of n directly to the non-zero child of n to produce B_{-n}.
   (b) Generate a DPO P_{-n} for B_{-n}.
   (c) Set $keep.node = 0$.
   (d) For each example e in E, do the following:
      i. For $i = 1$ to $|P|$,
      if p_i subsumes e, then set value(e) = leaf(p_i); break
      ii. For $i = 1$ to $|P_{-n}|$,
      if p_i subsumes e, then set newvalue(e) = leaf(p_i); break
      iii. If newvalue(e) $>$ value(e), set $keep.node = 1$; break
   (e) If $keep.node == 0$, set $B = B_{-n}$

Any node with $keep.node = 1$ must be kept because otherwise the value corresponding to some example will change. Hence we prune as much as is allowed by the example set. Again, if the example set is not rich enough, we may over-prune the FODD. R12-node is more sensitive than R12-edge. For instance, using the example set $\{\{p(1), q(1)\}, \{\neg p(1), q(1)\}\}$, R12-edge removes all edge redundancies in Figure 1(a) to yield Figure 1(b). With the same example set, however, R12-node will remove the non-redundant node $q(x)$. This is because the value of neither example changes by the removal of $q(x)$. For $q(x)$ to survive R12-node the example set must have an example like $\{\neg p(1), \neg q(1)\}$ demonstrating that the node is important.

Bootstrapping: Example Generation

The key to effective employment of R12-edge and R12-node is to provide these operators with a rich set of interesting examples. In the case of planning problems, the examples of interest are states visited during execution of the solution to a planning problem. Such states can be generated in a variety of ways. One potential method starts from a random state and runs episodes of simulated random walks through the state space. Another potential method employs a planner to solve a few sample planning problems and collect the states on the solution paths into the set of examples.

The methods we use in this paper start from a set of typical goal states and regress over ground actions to generate states from which the goal state is reachable. Regression from a ground state s over action a is possible only when a can achieve s from some state s'. Using STRIPS notation, this is easily verified by checking if s is subsumed by *PRECONDITIONS(a) + ADD-LIST(a) − DELETE-LIST(a)*. Such an s' is then generated by simply adding *DELETE-LIST(a)* to s and removing *ADD-LIST(a)* from s. This method of example generation is particularly suitable for SDP based systems like FODD-PLANNER because the same states are assigned new values by VI. In the following, we develop two variants of this approach both of which take a seed state s and regress from it.

Instance Regression (IR): We iteratively generate all possible states up to a certain specified depth using a BFS procedure. Regression over states at depth d produce states at depth $d + 1$ in the d^{th} iteration. The depth parameter can be set to the same value as the number of iterations of VI run by FODD-PLANNER because states from deeper levels

are not relevant to the value function. However, this method could generate a large example set eliminating the advantage of model-checking reductions. To mitigate this effect we introduce the following pruning techniques.

(1) Limit IR to use only those actions that bring about certain literals in the state. These literals are the preconditions of the previous action that generated this state through regression. In particular, if state s was regressed over action a' to produce state s' in the i^{th} iteration, then the preconditions of a' (which must be true in s') are maintained as special literals in the description of s'. When regressing over s' in iteration $i+1$, only those actions that bring about these special literals are considered. Thus we try to generate states that are further away from the goal.

(2) Identify and mark sub-goals so that they are not achieved more than once. For example, suppose $s_p \subset s$ are the special literals in s and as above we regress from s using a'. Mark literals in $s_p \cap$ *ADD-LIST*(a'). Now, in the $i+1^{th}$ iteration, when regressing from s', only those actions that generate states not containing the marked literals are considered. Considering the sequence of actions generated as a plan, this heuristic avoids re-achieving the same goal literal by the plan. Although incomplete, this heuristic is effective in limiting the number of states generated.

(3) Regress states over a composition of k actions instead of a single action. The parameter k is user defined. For example suppose action a is a composition of actions a'' and a'. Now, if s' regressed over a'' generates state s'', then s regressed over a generates s''. We add states s, s' and s'' to the set of examples. Regression continues from state s''. This technique avoids imposing the special and marked literal restrictions on every action and imposes them directly on compositions of actions, thereby allowing more freedom to search the state space in cases where the previous techniques are too restrictive.

Backward Random Walk (BRW): Instead of generating all states by iterative regression, as in IR, we run episodes of random walks backwards from the goal (sampling actions uniformly) without any of the above restrictions. The episode length and the number of episodes are user specified. This provides a varied set of states including states that are off the solution path for typical planning problems.

In practice we need a mix of examples generated by IR and BRW. To see why, let V be a value function that solves planning problem p optimally and S be a set of all possible states along any optimal solution path of p. Then V reduced by R12-edge against S is guaranteed to solve p optimally but V reduced by R12-node against S is not. As illustrated above, R12-node requires examples that gain value on removal of important nodes. Hence states off the solution path of p might be required for V to survive R12-node. The techniques with IR, however, are designed to generate only states along solution paths. Therefore in the example set we include all the states generated by IR and add states generates by BRW to yield a mixed set.

Conjunctive Goals and Goal Ordering

To enable efficient policy generation RMDP planners run VI with simple rewards (single literal goals). Then at plan execution time, action values for individual sub-goal literals in the conjunctive goal are combined. Sanner and Boutilier (2009) suggested the additive goal decomposition (AGD) heuristic of replacing a conjunctive goal with a sum of value functions for individual sub-goals and demonstrated that this works well on some problems. However, this heuristic is limited and does not apply well to many domains.

As an example consider a simple blocksworld scenario where 3 blocks a, b and c are on the table and the goal is to stack a on top of b on top of c. Clearly the block b has to be put on block c before a is put on b. However, the AGD heuristic adds up the values for each individual goal resulting in the effect that the action that puts a on b and the one that puts b on c get equal value. This results in the wrong action being chosen half of the time. For problems with a larger domain this can cause failure of plan execution.

To address this issue, we propose a new heuristic based on weighted goal ordering (WGO). The idea is to first get a partial ordering between goal literals. We use the following heuristic. For every pair of goal literals g_1 and g_2 we check if $\neg g_2$ is a precondition for some action to bring about g_1. If so, g_1 must be achieved before g_2 in the partial order. The intuition here is that preconditions expose some obvious ordering constraints on the goals. Identifying all such constraints amounts to solving a deterministic planning problem. Instead the heuristic identifies constraints that we can discover easily. Respecting this partial order we impose a total order on the goals in an arbitrary way. Finally the value of an action is calculated by adding the values of the individual goals literals as before, except that this time we weight the values of the individual sub-goals in proportion to their position in the total order. The value of the goal literal at position i is weighted by w^{i-1}. The weight parameter w ($0 < w \leq 1$) is user defined. As expected, small weights are better for domains with interacting sub-goals and large weights are better for domains where sub-goals are independent. In our example, an action that puts a on b will get a lower value than the action that puts b on c in the start state. These ideas help serialize any obvious ordered set of goals and gives a weak preference ordering on other sub-goals. This leads to significant improvements in performance in domains with interacting dependent sub-goals as we show in the next section.

Experiments on Planning Domains

In this section we present the results of our experiments on tireworld and blocksworld from IPC 2006, and boxworld from IPC 2008. Our intention here is to investigate:

(Q1) what are the contributions of the different parameters (number of training examples, number of iterations of VI, types of reductions, goal ordering heuristic) of the self-taught model-checking system, and **(Q2)** whether our self-taught model-checking system can effectively speed up decision theoretic planning while matching performance with the theorem proving system and other state of the art systems in stochastic planning.

To this aim, we generated a value function for each domain by running the FODD-PLANNER in three different configurations based on the method used to reduce FODDs: **FODD-PL** (theorem proving reductions only (Joshi and

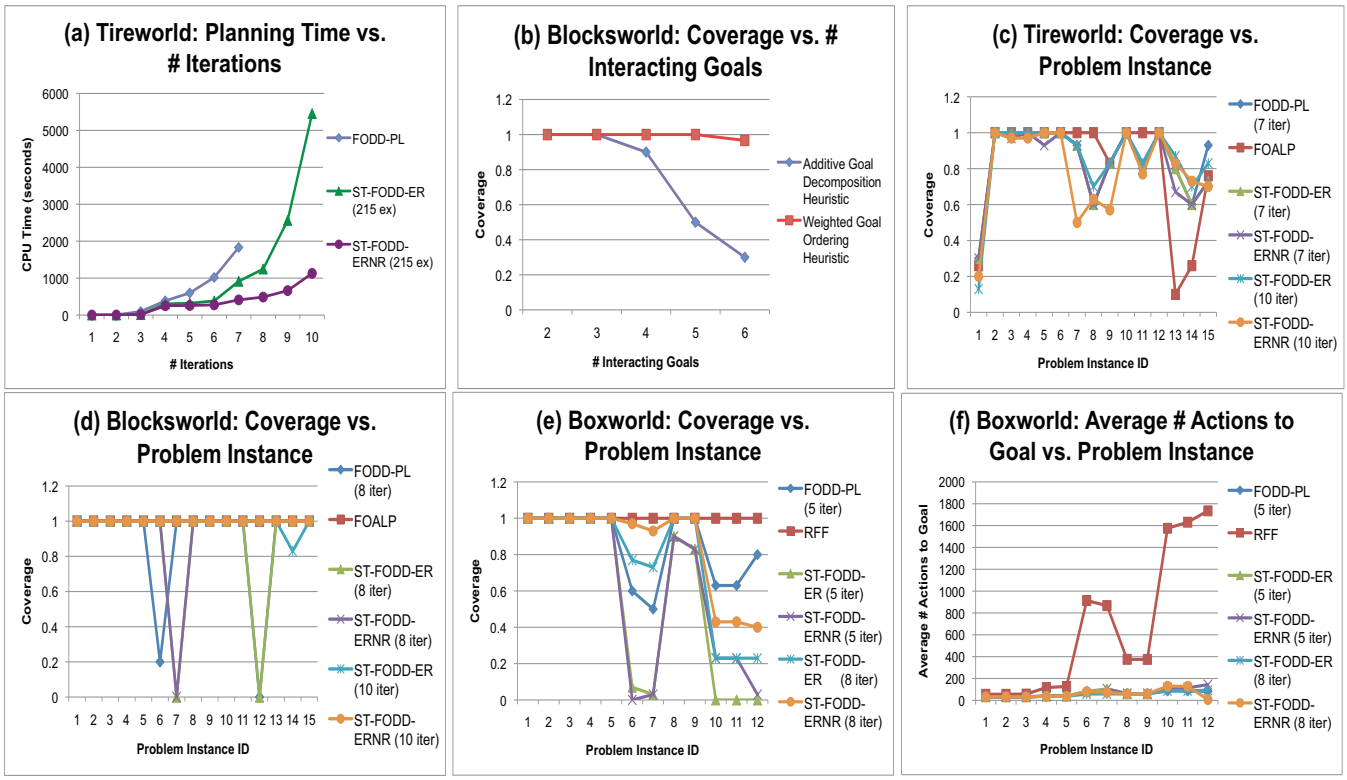

Figure 2: Summary of Experimental Results.

Khardon 2008)), **ST-FODD-ER** (R12 edge removal reduction and a theorem proving node removal reduction R11 (Joshi and Khardon 2008)), and **ST-FODD-ERNR** (R12 edge and node removal reductions only). To measure performance we follow IPC standards: The methods are compared in terms of coverage, i.e., percentage of problems solved over 30 rounds, average plan length, and cpu time. We generated examples by mixing states generated by the IR and BRW methods as explained above. In keeping with the IPC norms, we limited the offline planning time. We did not limit the plan execution time but set a limit of 200 on the plan length counting plans that violated the limit as failed. Further details are given below.

Timeout Mechanism

Before investigating **(Q1)** and **(Q2)** we must address one further practical aspect of the system. Since we switched from theorem proving to model-checking, the cost of subsumption tests heavily affects system run time. Subsumption problems show a phase transition phenomenon (Maloberti and Sebag 2004) and, while most problems are easy, certain types of problems have a very high cost. In preliminary experiments on tireworld, we found that while over 99% of subsumption tests ran within 50 milliseconds, a very small number took over 30 minutes each. To avoid these costly tests we use a timeout mechanism within the subsumption routine. Subsumption simply fails if it runs beyond a specified time limit. Statistics taken over different domains and iterations showed that a time limit of 1 second avoids most of the computational burden and that one cannot get further significant speedup by lowering this threshold. We omit details of these experiments due to space constraints. Importantly, the following experiments show that timeout achieves both speedup and good performance of the planning system.

(Q1) System Characteristics

We start by characterizing scaling and performance. The model-checking reductions use training examples, and therefore, similar to standard machine learning more examples should give better performance. This can be demonstrated by increasing training set size and measuring performance. Our experiments show, for example, that the performance of ST-FODD-ER (for the tireworld domain, 7 iterations) converges at about 200 examples. They also show that the run time increases roughly linearly with training set size. Details of these results are omitted due to space constraints. Figure 2(a) illustrates scaling by showing the dependence of runtime on the number of iterations of VI for the 3 methods where ST-FODD-ER and ST-FODD-ERNR use a fixed training set of 215 examples. We observe significant speedup where ST-FODD-ERNR is more efficient than ST-FODD-ER, which is more efficient than FODD-PL. The speedup allows us to run more iterations in reasonable run times; for example running 10 iterations of FODD-PL requires more than 2 days (not shown in graph) whereas ST-FODD-ER requires about 1.5 hours and ST-FODD-ERNR requires less than 0.5 hour.

We next illustrate the performance of the goal ordering

M	PL	ER	ERNR	ER	ERNR
I	8	8	8	10	10
C	12465.95	222.33	78.69	865.57	357.68

Table 1: Blocksworld: Planning time taken in CPU seconds (C) by methods (M) FODD-PL (PL), ST-FODD-ER (ER) and ST-FODD-ERNR (ERNR) for iterations (I)

M	PL	ER	ERNR	ER	ERNR
I	5	5	5	8	8
C	3332.33	302.8	117.7	3352.93	618.85

Table 2: Boxworld: Planning time taken in CPU seconds (C) by methods (M) FODD-PL (PL), ST-FODD-ER (ER) and ST-FODD-ERNR (ERNR) for iterations (I)

heuristic. To explore this space, we generated a sequence of blocksworld problems with an increasing number of interacting goals. Figure 2(b) illustrates this behavior with a plot of the percentage of planning problems solved by ST-FODD-ER against the number of interacting goal literals for a family of problems all with 5 blocks where for WGO we set $w = 0.8$. Both AGD and WGO heuristics do well on problems with up to 3 interacting goals. However, beyond 3, the performance of AGD degrades while WGO is still successful. The performance curves are similar for FODD-PL and ST-FODD-ERNR across iterations (details omitted).

(Q2) Tireworld

This domain is from IPC 2006. All problems have single literal goals. Hence goal combining heuristics are not needed.

We generated training examples by running IR with the d parameter set to 10 and the k parameter set to 1 and running BRW for 10 episodes of length 20. The seed state for both methods was designed so as to have a map of 10 locations connected linearly. The vehicle is in the first location. While our methods are not very sensitive to the choice of seed state, using a constrained scenario helps in limiting the number of examples generated. Step 2 of the VI algorithm above requires adding of deterministic action alternative FODDs at each iteration. These FODDs must be standardized apart to guarantee correctness; following Joshi and Khardon (2008) we relax this requirement and do not standardize apart after the third iteration.

Runtime and speedup for this domain were discussed above. For all the settings in Figure 2(a) offline planning and execution together completed in less than the IPC limit of 4 hours. As shown in Figure 2(c) ST-FODD-ER and ST-FODD-ERNR achieve the same level of coverage as FODD-PL and comparable to FOALP (Sanner and Boutilier 2009), one of the top ranking systems from IPC 2006. Results for plan length (figure omitted) show comparable results to FODD-PL which is slightly better than other systems.

(Q2) Blocksworld

In IPC 2006 this domain was described by 7 probabilistic actions. For the purpose of VI we determinized the domain by replacing every probabilistic action by its most probable deterministic alternative. The resultant domain consists of 4 deterministic actions, pick-up-block-from-table, pick-up-block-from-block, put-block-on-table, put-block-on-block.

IPC posted 15 problems with varying degrees of difficulty. The goal consists of multiple interacting goal literals. We generated training examples using both IR and BRW. For IR the d parameter was set to 15 and k was set to 2. For BRW we ran 10 episodes of length 20. The seed example was hand constructed to be a state satisfying the (single literal) goal and, in addition, a tower including all other blocks.

Table 1 shows that ST-FODD-ER and ST-FODD-ERNR achieve respectively 56 and 158 fold speedup in terms of offline planning time over FODD-PL at 8 iterations. Execution times varied by problem. The harder problems took half an hour per round. The easier problems (1 to 10) took less than 75 seconds per round. The longer execution times are a result of having to solve a large number of subsumption tests for action selection. Once again we observed ST-FODD-ER and ST-FODD-ERNR achieve the same level of performance as FODD-PL and FOALP in terms of coverage (Figure 2(d)) and plan length (figure omitted).

(Q2) Harder Problems

Increasing the number of iterations of VI with ST-FODD-ER and ST-FODD-ERNR does not improve performance on the IPC problems. To exhibit such a phenomenon we generated harder problems for the tireworld and blocksworld domains. Interestingly, for tireworld we used the IPC problem generator but could not find hard problems that could demonstrate the difference. This shows that the generator produced relatively easy problems. It is easy to describe such examples where the goal is farther away and where at every state we have several irrelevant but applicable actions. In this case a shallow value function essentially chooses actions randomly. We constructed such examples (details omitted) where 10 iterations of ST-FODD-ER achieves 100% coverage as opposed to 3% with 7 iterations of FODD-PL. This shows a significant advantage of the new methods.

For blocksworld, again the ability to run more iterations pays off when solving harder problems. It is easy to generate problems showing this difference. Such problems require long sequence of actions to the goal and a large number of possible but irrelevant actions at every step so that policies based on a shallow value functions are forced to take random actions and fail. We generated such a problem where 8 iterations of FODD-PL achieves only 50% coverage in contrast with full coverage with 15 iterations of ST-FODD-ERNR. Details are omitted due to space constraints.

(Q2) Boxworld

This domain is from IPC 2008. For the purpose of VI we determinized this domain making the only probabilistic action *drive* deterministic. IPC posted 15 problems with varying levels of difficulty for this domain. Competition results show that RFF (Teichteil-Koenigsbuch, Infantes, and Kuter 2008) was the *only* system that solved any of the 15 problems. RFF could not solve problems 13 to 15. Hence we omit results for those.

Training examples were generated using both IR and BRW. For IR we set the d parameter to 8 and k parameter to 1. For BRW we ran 10 episodes of length 20. The seed example was designed to have a map of 10 cities connected linearly. Each location connects to 2 neighbors by a two way road except the locations at the extremes which connect to only one location; 2 boxes, 2 trucks and a plane were placed in the 4^{th} city. Unlike blocksworld, the goals in this domain do not interact and the WGO heuristic returns no relative order between goal literals. Therefore we set w to 1.0 indicating that there is no order in achieving the goals. This setting reduces the WGO heuristic to the AGD heuristic.

Table 2 shows that ST-FODD-ER and ST-FODD-ERNR respectively achieve 11 and 28 fold speed up over FODD-PL at 5 iterations. Execution times varied from 100 seconds per round on the easier problems to 1.5 hours per round on hard problems. These long execution times might explain why other planning systems failed at the IPC where a strict time bound on execution was used. Figure 2(e) and (f) show performance in terms of coverage and plan length. We observe that additional iterations are crucial in this domain. With 8 iterations ST-FODD-ER and ST-FODD-ERNR achieve full or near full coverage on all problems except 10, 11 and 12. In comparison RFF achieves full coverage on all problems. On the other hand our system achieves significantly better plan lengths than RFF even on problems where full coverage was achieved. Results for average reward (figure omitted) show comparable values for the two systems.

Conclusion

In this paper, we presented a new paradigm for decision theoretic planning by learning. This paradigm allows an agent to speed up Symbolic Dynamic Programming significantly by using a set of reference states (but without any explicit information about optimal actions or values for these states). Self-taught or provided by a mentor, the information gleaned from these states is used to remove any complex specifications within the value function that are irrelevant to the given states, thereby focusing on the region of interest.

We demonstrate this idea in the context of the RMDP system FODD-PLANNER that uses FODDs to represent the value function compactly. By extending and employing model-checking reductions for FODDs (Joshi, Kersting, and Khardon 2009), we vastly improve efficiency of planning. We introduce novel methods for auto-generation of states of interest and heuristics for combining goals during plan execution. Experiments on domains and problems from the IPC suggest that this technique not only greatly improves the efficiency of the planning system, but also allows it to solve harder planning problems.

Although all experiments and demonstrations have been in the context of FODDs and the FODD-PLANNER system, we believe that self-taught planning can also be applied to other SDP based solvers of RMDPs. This is an interesting direction for future work. Further research is required to improve the runtime of the execution module. This is partly a question of optimizing code and partly inherent in using a complex policy. Ideas used in the context of production rule systems may be useful here. Another promising idea is to acquire the training examples from multiple teachers each specializing in a separate (but possibly overlapping) part of the state space, similar to (Price and Boutilier 2003). In contrast with behavioral cloning e.g. (Morales and Sammut 2004) where the performance typically degrades if the learner gets contradicting examples from multiple teachers, our approach handles this case in a natural way.

Acknowledgments

Kristian Kersting was supported by the Fraunhofer ATTRACT fellowship STREAM. Saket Joshi and Roni Khardon were partly supported by NSF grant IIS 0936687.

References

Boutilier, C.; Reiter, R.; and Price, B. 2001. Symbolic dynamic programming for first-order mdps. In *Proceedings of IJCAI*, 690–700.

Fern, A.; Yoon, S.; and Givan, R. 2003. Approximate policy iteration with a policy language bias. In *Proceedings of NIPS*.

Gretton, C., and Thiebaux, S. 2004. Exploiting first-order regression in inductive policy selection. In *Proceedings of UAI*.

Groote, J., and Tveretina, O. 2003. Binary decision diagrams for first order predicate logic. *Journal of Logic and Algebraic Programming* 57:1–22.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. Spudd: Stochastic planning using decision diagrams. In *Proceedings of UAI*, 279–288.

Hölldobler, S.; Karabaev, E.; and Skvortsova, O. 2006. FluCaP: a heuristic search planner for first-order MDPs. *JAIR* 27:419–439.

Joshi, S., and Khardon, R. 2008. Stochastic planning with first order decision diagrams. In *Proceedings of ICAPS*.

Joshi, S.; Kersting, K.; and Khardon, R. 2009. Generalized first order decision diagrams for first order markov decision processes. In *Proceedings of IJCAI*.

Kersting, K.; Otterlo, M. V.; and De Raedt, L. 2004. Bellman goes relational. In *Proceedings of ICML*.

Lloyd, J. 1987. *Foundations of Logic Programming*. Springer Verlag. Second Edition.

Maloberti, J., and Sebag, M. 2004. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning* 55:137–174.

Morales, E., and Sammut, C. 2004. Learning to fly by combining reinforcement learning with behavioral cloning. In *Proceedings of the ICML*.

Price, B., and Boutilier, C. 2003. Accelerating reinforcement learning through implicit imitation. *JAIR* 19:569–629.

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

Sanner, S., and Boutilier, C. 2009. Practical solution techniques for first order mdps. *AIJ* 173:748–788.

Teichteil-Koenigsbuch, F.; Infantes, G.; and Kuter, U. 2008. Rff: A robust ff-based mdp planning algorithm for generating policies with low probability of failure. In *Sixth IPC at ICAPS*.

Wang, C.; Joshi, S.; and Khardon, R. 2008. First order decision diagrams for relational mdps. *JAIR* 31:431–472.

Yoon, S.; Fern, A.; and Givan, R. 2007. Ff-replan: A baseline for probabilistic planning. In *Proceedings of ICAPS*, 352–.