Symbolic Dynamic Programming

Scott Sanner
NICTA, Statistical Machine Learning Group, Canberra, Australia

Kristian Kersting
Fraunhofer IAIS, Dept. of Knowledge Discovery, Sankt Augustin, Germany

Synonyms

Relational Dynamic Programming, Dynamic Programming for Relational Domains, Relational Value Iteration

Definition

Symbolic dynamic programming is a generalization of the dynamic programming technique for solving Markov decision processes that exploits the symbolic structure in the solution of relational and first-order logical Markov decision processes through a lifted version of dynamic programming.

Motivation and Background

Decision-theoretic planning aims at constructing a policy for acting in an uncertain environment that maximizes an agent's expected utility along a sequence of steps. For this task, Markov decision processes (MDPs) have become the standard model. However, classical dynamic programming algorithms for solving MDPs require explicit state and action enumeration, which is often impractical: the number of states and actions grows very quickly with the number of domain objects and relations. In contrast, symbolic dynamic programming (SDP) algorithms seek to avoid explicit state and action enumeration through the symbolic representation of an MDP and a corresponding symbolic derivation of its solution, such as a value function. In essence, SDP algorithms exploit the symbolic structure of the MDP representation to construct a minimal logical partition of the state space required to make all necessary value distinctions.

Theory and Solution

Consider an agent acting in a simple variant of the BoxWorld problem. There are several cities such as *london*, *paris* etc., trucks $truck_1$, $truck_2$ etc., and boxes box_1, box_2 etc. The agent can load a box onto a truck or unload it and can drive a truck from one city to another. Only when a particular box, say box box_1, is in a particular city, say *paris*, the agent receives a positive reward. The agent's learning task is now to find a policy for action selection that maximizes its reward over the long term.

A great variety of techniques for solving such decision-theoretic planning tasks have been developed over the last decades. Most of them assume atomic representations, which essentially amounts to enumerating all unique configurations of trucks, cities, and boxes. It might then be possible to learn, for example, that taking action *action234* in state *state42* is worth 6.2 and leads to state *state654321*. Atomic representations are simple and learning can be implemented using simple look-up tables. These look-up tables, however, can be intractably large as atomic representations easily explode. Furthermore, they do not easily generalize across different numbers of domain objects.[1]

In contrast, symbolic dynamic programming (SDP) assumes a relational or first-order logical representation of an MDP (as given in Fig. 1) to exploit the existence of domain objects, relations over these objects, and the ability to express objectives and action effects using quantification. It is then possible to learn that to get box b to *paris*, the agent drives a truck to the city of b, loads box_1 on the truck, drives the truck to *paris*, and finally unloads the box box_1 in *paris*. This is essentially encoded in the symbolic value function shown in Fig. 2, which was computed by discounting rewards t time steps into the future by 0.9^t. The key features to note here are the state and action abstraction in the value and policy representation that are afforded by the first-order specification and solution of the problem. That is, this solution does not refer to any specific set of domain objects, such as $City = \{paris, berlin, london\}$, but rather it provides a solution for *all possible domain object instantiations.* And while classical dynamic programming techniques could never solve these problems for large domain instantiations (since they would have to enumerate all states and actions), a domain-independent SDP solution to this particular problem is quite simple due to the power of state and action abstraction.

[1] We use the term *domain* in the first-order logical sense of an object universe. The term should not be confused with a planning *problem* such as BoxWorld or BlocksWorld.

Figure 1: A formal desciption of the BoxWorld adapted from [1]. We use a simple STRIPS [2] add and delete list representation of actions and, as a simple probabilistic extension in the spirit of PSTRIPS [3], we assign probabilities that an action successfully executes conditioned on various state properties.

- *Domain Object Types (i.e., sorts)*: Box, Truck, $\mathit{City} = \{\mathit{paris}, \ldots\}$
- *Relations (with parameter sorts)*:
  $\mathit{BoxIn}(\mathit{Box}, \mathit{City})$, $\mathit{TruckIn}(\mathit{Truck}, \mathit{City})$, $\mathit{BoxOn}(\mathit{Box}, \mathit{Truck})$
- *Reward*: if $\exists b.\mathit{BoxIn}(b, \mathit{paris})$ 10 else 0
- *Actions (with parameter sorts)*:
  - $\mathit{load}(\mathit{Box} : b, \mathit{Truck} : t, \mathit{City} : c)$:
    - Success Probability: if $(\mathit{BoxIn}(b, c) \wedge \mathit{TruckIn}(t, c))$ then .9 else 0
    - Add Effects on Success: $\{\mathit{BoxOn}(b, t)\}$
    - Delete Effects on Success: $\{\mathit{BoxIn}(b, c)\}$
  - $\mathit{unload}(\mathit{Box} : b, \mathit{Truck} : t, \mathit{City} : c)$:
    - Success Probability: if $(\mathit{BoxOn}(b, t) \wedge \mathit{TruckIn}(t, c))$ then .9 else 0
    - Add Effects on Success: $\{\mathit{BoxIn}(b, c)\}$
    - Delete Effects on Success: $\{\mathit{BoxOn}(b, t)\}$
  - $\mathit{drive}(\mathit{Truck} : t, \mathit{City} : c_1, \mathit{City} : c_2)$:
    - Success Probability: if $(\mathit{TruckIn}(t, c_1))$ then 1 else 0
    - Add Effects on Success: $\{\mathit{TruckIn}(t, c_2)\}$
    - Delete Effects on Success: $\{\mathit{TruckIn}(t, c_1)\}$
  - *noop*
    - Success Probability: 1
    - Add Effects on Success: $\emptyset$
    - Delete Effects on Success: $\emptyset$

Background: Markov Decision Processes (MDPs)

In the MDP [4] model, an agent is assumed to fully observe the current state and choose an action to execute from that state. Based on that state and

Figure 2: A decision-list representation of the optimal policy and expected discounted reward value for the BoxWorld problem.

if $(\exists b.\mathit{BoxIn}(b, \mathit{paris}))$ then do *noop* (value = 100.00)

else if $(\exists b, t.\mathit{TruckIn}(t, \mathit{paris}) \wedge \mathit{BoxOn}(b, t))$ then do $\mathit{unload}(b, t)$ (value = 89.0)

else if $(\exists b, c, t.\mathit{BoxOn}(b, t) \wedge \mathit{TruckIn}(t, c))$ then do $\mathit{drive}(t, c, \mathit{paris})$ (value = 80.0)

else if $(\exists b, c, t.\mathit{BoxIn}(b, c) \wedge \mathit{TruckIn}(t, c))$ then do $\mathit{load}(b, t)$ (value = 72.0)

else if $(\exists b, c_1, t, c_2.\mathit{BoxIn}(b, c_1) \wedge \mathit{TruckIn}(t, c_2))$ then do $\mathit{drive}(t, c_2, c_1)$ (value = 64.7)

else do *noop* (value = 0.0)

action, nature then chooses a next state according to some fixed probability distribution. In an infinite-horizon MDP, this process repeats itself indefinitely. Assuming there is a reward associated with each state and action, the goal of the agent is to maximize the expected sum of discounted rewards received over an infinite horizon.[2] This criterion assumes that a reward received t steps in the future is discounted by γ^t where γ is a discount factor satisfying $0 \leq \gamma < 1$. The goal of the agent is to choose its actions in order to maximize the expected, discounted future reward in this model.

Formally, a finite state and action MDP is a tuple: $\langle S, A, T, R \rangle$ where: S is a finite state space, A is a finite set of actions, T is a transition function: $T : S \times A \times S \rightarrow [0, 1]$ where $T(s, a, \cdot)$ is a probability distribution over S for any $s \in S$ and $a \in A$, and R is a bounded reward function $R : S \times A \rightarrow \mathbb{R}$.

As stated above, our goal is to find a policy that maximizes the infinite horizon, discounted reward criterion: $E_\pi[\sum_{t=0}^{\infty} \gamma^t \cdot r^t | s_0]$, where r^t is a reward obtained at time t, γ is a discount factor as defined above, π is the policy being executed, and s_0 is the initial starting state. Based on this reward criterion, we define the value function for a policy π as the following:

$$V_\pi(s) = E_\pi \left[\sum_{t=0}^{\infty} \gamma^t \cdot r^t \;\middle|\; s_0 = s \right] \tag{1}$$

Intuitively, the value function for a policy π is the expected sum of discounted rewards accumulated while executing that policy when starting from state s.

[2] Although we do not discuss it here, there are other alternatives to discounting such as averaging the reward received over an infinite horizon.

For the MDP model discussed here, the optimal policy can be shown to be stationary [4]. Consequently, we use a stationary policy representation of the form $\pi : S \rightarrow A$, with $\pi(s)$ denoting the action to be executed in state s. An optimal policy π^* is the policy that maximizes the value function for all states. We denote the optimal value function over an indefinite horizon as $V^*(s)$ and note that it satisfies the following equality:

$$V^*(s) = \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \cdot V^*(s') \right\} \tag{2}$$

Bellman's *principle of optimality* [5] establishes the following relationship between the optimal value function $V^t(s)$ with a finite horizon of t steps remaining and the optimal value function $V^{t-1}(s)$ with a finite horizon of $t-1$ steps remaining:

$$V^t(s) = \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') \cdot V^{t-1}(s') \right\} \tag{3}$$

A *dynamic programming* approach for computing the optimal value function over an indefinite horizon is known as value iteration and directly implements Eq. 3 to compute Eq. 1 by successive approximation. As sketched in Fig. 3, we start with arbitrary $V^0(s)$ (e.g., $\forall s\, V^0(s) = 0$) and perform the Bellman backup given in Eq. 3 for each state $V^1(s)$ using the value of $V^0(s)$. We repeat this process for each t to compute $V^t(s)$ from the memoized values for $V^{t-1}(s)$ until we have computed the intended t-stages-to-go value function. $V^t(s)$ will converge to $V^*(s)$ as $t \rightarrow \infty$ [4].

Often, the Bellman backup is rewritten in two steps to separate out the action regression and maximization steps. In this case, we first compute the t-stages-to-go Q-function for every action and state:

$$Q^t(s,a) = R(s,a) + \gamma \cdot \sum_{s' \in S} T(s,a,s') \cdot V^{t-1}(s') \tag{4}$$

Then we maximize over each action to determine the value of the regressed state:

$$V^t(s) = \max_{a \in A} \left\{ Q^t(s,a) \right\} \tag{5}$$

This is clearly equivalent to Eq. 3 but is in a form that we will refer to later since it separates the algorithm into its two conceptual components: decision-theoretic regression and maximization.

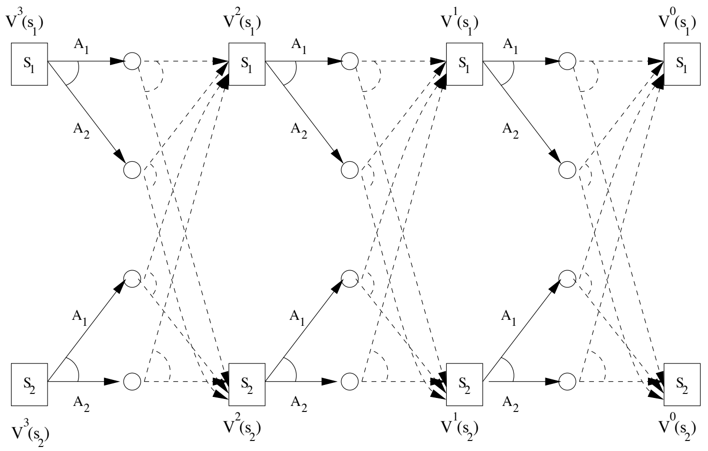

Figure 3: A diagram demonstrating a *dynamic programming* regression-based evaluation of the MDP value function. Dashed lines are used in the expectation computation of the Q-function: for each action, take the expectation over the values of possible successor states. Solid lines are used in the max computation: determine the highest valued action to take in each state.

First-order Markov Decision Processes

A first-order MDP (FOMDP) can be thought of as a universal MDP that abstractly defines the state, action, transition, and reward tuple $\langle S, A, T, R\rangle$ for an infinite number of ground MDPs. To make this idea more concrete, consider the BoxWorld problem defined earlier. While we have not yet formalized the details of the FOMDP representation, it should be clear that the BoxWorld dynamics hold for any instantiation of the domain objects: *Box*, *Truck*, and *City*. For instance, assume the domain instantiation consists of two boxes $\mathit{Box} = \{\mathit{box}_1, \mathit{box}_2\}$, two trucks $\mathit{Truck} = \{\mathit{truck}_1, \mathit{truck}_2\}$ and two cities $\mathit{City} = \{\mathit{paris}, \mathit{berlin}\}$. Then the resulting ground MDP has 12 state-variable atoms (each atom being *true* or *false* in a state), four atoms for *BoxIn* such as $\mathit{BoxIn}(\mathit{box}_1, \mathit{paris})$, $\mathit{BoxIn}(\mathit{box}_2, \mathit{paris}), \ldots$, four atoms for *TruckIn* such as $\mathit{TruckIn}(\mathit{truck}_2, \mathit{paris}), \ldots$ and four atoms for *BoxOn* such as $\mathit{BoxOn}(\mathit{box}_2, \mathit{truck}_1), \ldots$. There are also 24 possible actions (eight for each of *load*,*unload*, and *drive*) such as $\mathit{load}(\mathit{box}_1, \mathit{truck}_1, \mathit{paris})$, $\mathit{load}(\mathit{box}_1, \mathit{truck}_1, \mathit{berlin})$, $\mathit{drive}(\mathit{truck}_2, \mathit{paris}, \mathit{paris})$, $\mathit{drive}(\mathit{truck}_2, \mathit{paris}, \mathit{berlin})$, etc. where the transition function directly follows from the ground instantions of the corresponding PSTRIPS operators. The reward function looks like: if $(\mathit{BoxIn}(\mathit{box}_1, \mathit{paris}) \vee \mathit{BoxIn}(\mathit{box}_2, \mathit{paris}))$ 10 else 0.

Therefore, to solve a FOMDP, we could ground it for a specific domain instantiation to obtain a corresponding ground MDP. Then we could apply classical MDP solution techniques to solve this ground MDP. However, the obvious drawback to this approach is that the number of state variables and actions in the ground MDP grow at least linearly as the domain size increases. And even if the ground MDP could be represented within memory constraints, the number of distinct ground states grows exponentially with the number of state variables, thus rendering solutions that scale with state size intractable even for moderately small numbers of domain objects.

An alternative idea to solving a FOMDP at the ground level is to solve the FOMDP directly at the first-order level using symbolic dynamic programming, thereby obtaining a solution that applies universally to all possible domain instantiations. While the exact representation and symbolic dynamic programming solution of FOMDPs differs among the variant formalisms, they all share the same basic first-order representation of rewards, probabilities, and values that we outline next. To highlight this, we introduce a graphical *case notation* to allow first-order specifications of the rewards, probabilities, and values required for FOMDPs:

$$\mathit{case} = \begin{array}{|c|}\hline \phi_1 : t_1 \\ \hline : \quad : \; : \\ \hline \phi_n : t_n \\ \hline \end{array}$$

Here the ϕ_i are *state formulae* and the t_i are terms. Often the t_i will be constants and the ϕ_i will partition state space. To make this concrete, we represent our BoxWorld FOMDP reward function as the following *rCase* statement:

$$\mathit{rCase} = \begin{array}{|lr|}\hline \exists b.\mathit{BoxIn}(b, \mathit{paris}) & : 10 \\ \hline \neg\exists b.\mathit{BoxIn}(b, \mathit{paris}) : & 0 \\ \hline \end{array}$$

Here we see that the first-order formulae in the case statement divide all possible ground states into two regions of constant-value: when there exists a box in Paris, a reward of 10 is achieved, otherwise a reward of 0 is achieved. Likewise the value function *vCase* that we derive through symbolic dynamic programming can be represented in exactly the same manner. Indeed, as we will see shortly, $\mathit{vCase}^0 = \mathit{rCase}$ in the first-order version of value iteration.

To state the FOMDP transition function for an action, we decompose stochastic "agent" actions into a *collection* of deterministic actions, each corresponding to a possible outcome of the stochastic action. We then specify a distribution according to which "nature" may choose a deterministic action from this set whenever the stochastic action is executed.

Letting $A(\vec{x})$ be a stochastic action with nature's choices (i.e., deterministic actions) $n_1(\vec{x}), \cdots, n_k(\vec{x})$, we represent the distribution over $n_i(\vec{x})$ given $A(\vec{x})$ using the notation $pCase(n_j(\vec{x}), A(\vec{x}))$. Continuing our logistics example, if the success of driving a truck depends on whether the destination city is *paris* (perhaps due to known traffic delays), then we decompose the stochastic *drive* action into two deterministic actions *driveS* and *driveF*, respectively denoting success and failure. Then we can specify a distribution over nature's choice deterministic outcome for this stochastic action:

$$\begin{array}{rl} pCase(& driveS(t, c_1, c_2), \\ & drive(t, c_1, c_2)) \end{array} \quad = \quad \begin{array}{|c|} \hline c_2 = paris : 0.6 \\ \hline c_2 \neq paris : 0.9 \\ \hline \end{array}$$

$$\begin{array}{rl} pCase(& driveF(t, c_1, c_2), \\ & drive(t, c_1, c_2)) \end{array} \quad = \quad \begin{array}{|c|} \hline c_2 = paris : 0.4 \\ \hline c_2 \neq paris : 0.1 \\ \hline \end{array}$$

Intuitively, to perform an operation on case statements, we simply perform the corresponding operation on the intersection of all case partitions of the operands. Letting each ϕ_i and ψ_j denote generic first-order formulae, we can perform the "cross-sum" $\oplus$ of case statements in the following manner:

$$\begin{array}{|c|} \hline \phi_1 : 10 \\ \hline \phi_2 : 20 \\ \hline \end{array} \quad \oplus \quad \begin{array}{|c|} \hline \psi_1 : 1 \\ \hline \psi_2 : 2 \\ \hline \end{array} \quad = \quad \begin{array}{|c|} \hline \phi_1 \wedge \psi_1 : 11 \\ \hline \phi_1 \wedge \psi_2 : 12 \\ \hline \phi_2 \wedge \psi_1 : 21 \\ \hline \phi_2 \wedge \psi_2 : 22 \\ \hline \end{array}$$

Likewise, we can perform $\ominus$, $\otimes$, and max operations by, respectively, subtracting, multiplying, or taking the max of partition values (as opposed to adding them) to obtain the result. Some partitions resulting from the application of the $\oplus$, $\ominus$, and $\otimes$ operators may be inconsistent; we simply discard such partitions (since they can obviously never correspond to any world state).

We define another operation on case statements $\max \exists \vec{x}$ that is crucial for symbolic dynamic programming. Intuitively, the meaning of $\max \exists \vec{x}\, case(\vec{x})$ is a case statement where the maximal value is assigned to each region of state space where there exists a satisfying instantiation of $\vec{x}$. To make these ideas concrete, following is an exposition of how the $\max \exists \vec{x}$ may be explicitly computed:

$$\max \exists \vec{x} \;\; \begin{array}{|c|} \hline \psi_1(\vec{x}) : 1 \\ \hline \psi_2(\vec{x}) : 2 \\ \hline \psi_3(\vec{x}) : 3 \\ \hline \end{array} \;\; = \;\; \begin{array}{|ll|} \hline \exists \vec{x}\, \psi_3(\vec{x}) & : 3 \\ \hline \neg(\exists \vec{x}\, \psi_3(\vec{x})) \wedge \exists \vec{x}\, \psi_2(\vec{x}) & : 2 \\ \hline \neg(\exists \vec{x}\, \psi_3(\vec{x})) \wedge \neg(\exists \vec{x}\, \psi_2(\vec{x})) \wedge \exists \vec{x}\, \psi_1(\vec{x}) & : 1 \\ \hline \end{array}$$

Here we have simply sorted partitions in order of value and have ensured that the highest value is assigned to partitions in which there exists a satisfying instantiation of $\vec{x}$ by rendering lower value partitions disjoint from their higher-value antecedents.

Symbolic Dynamic Programming

Symbolic dynamic programming (SDP) is a dynamic programming solution to FOMDPs that produces a logical case description of the optimal value function. This is achieved through the operations of first-order decision-theoretic regression and symbolic maximization that perform the traditional dynamic programming Bellman backup at an abstract level without explicit enumeration of either the state or action spaces of the FOMDP. Among many uses, the application of SDP leads to a domain-independent value iteration solution to FOMDPs.

Suppose we are given a value function in the form *vCase*. The first-order decision-theoretic regression (FODTR) [1] of this value function through an action $A(\vec{x})$ yields a case statement containing the logical description of states and values that would give rise to *vCase* after doing action $A(\vec{x})$. This is analogous to classical goal regression, the key difference being that action $A(\vec{x})$ is stochastic. In MDP terms, the result of FODTR is a case statement representing a Q-function.

We define the *first-order decision theoretic regression (FODTR)* operator in the following manner:

$$\begin{aligned} \mathit{FODTR}[\mathit{vCase}, A(\vec{x})] =& \mathit{rCase} \oplus \\ & \gamma\ [\oplus_j \{ \mathit{pCase}(n_j(\vec{x})) \otimes \mathit{Regr}[\mathit{vCase}, A(\vec{x})] \}] \end{aligned} \tag{6}$$

Note that we have not yet defined the regression operator $\mathit{Regr}[\mathit{vCase}, A(\vec{x})]$. As it turns out, the implementation of this operator is specific to a given FOMDP language and SDP implementation. We simply remark that the regression of a formula ψ through an action $A(\vec{x})$ is a formula ψ' that holds prior to $A(\vec{x})$ being performed iff ψ holds after $A(\vec{x})$. However, regression is a deterministic operator and thus FODTR takes the expectation of the regression over all possible outcomes of a stochastic action according to their respective probabilities.

It is important to note that the case statement resulting from FODTR contains free variables for the action parameters $\vec{x}$. That is, for any constant binding $\vec{c}$ of these action parameters such that $\vec{x} = \vec{c}$, the case statement specifies a well-defined logical description of the value that can be obtained by taking action $A(\vec{c})$ and following a policy so as to obtain the value given by *vCase* thereafter. However, what we really need for symbolic dynamic programming is a logical description of a Q-function that tells us the highest value that can be achieved for *any* action instantiation. This leads us to the following $qCase(A(\vec{x}))$ definition of a first-order Q-function that makes use of the previously defined $\max \exists \vec{x}$ operator:

$$qCase^t(A(\vec{x})) = \max \exists \vec{x}.\, FODTR[vCase^{t-1}, A(\vec{x})] \tag{7}$$

Intuitively, $qCase^t(A(\vec{x}))$ is a logical description of the Q-function for action $A(\vec{x})$ indicating the best value that could be achieved by *any* instantiation of $A(\vec{x})$. And by using the case representation and action quantification in the $\max \exists \vec{x}$ operation, FODTR effectively achieves *both* action and state abstraction.

At this point, we can regress the value function through a *single* action, but to complete the dynamic programming step, we need to know the maximum value that can be achieved by *any* action (e.g., in the BoxWorld FOMDP, our possible action choices are $unload(b,t,c)$, $load(b,t,c)$, and $drive(t,c_1,c_2)$). Fortunately, this turns out to be quite easy. Assuming we have m actions $\{A_1(\vec{x}_1), \ldots, A_m(\vec{x}_m)\}$, we can complete the SDP step in the following manner using the previously defined max operator:

$$vCase^t = \max_{a \in \{A_1(\vec{x}_1), \ldots, A_m(\vec{x}_m)\}} qCase^t(a) \tag{8}$$

While the details of SDP may seem very abstract at the moment, there are several examples for specific FOMDP languages that implement SDP as described above, for which we provide references below. Nonetheless, one should note that the SDP equations given here are exactly the "lifted" versions of the traditional dynamic programming solution to MDPs given previously in Eqs. 4 and 5. The reader may verify — on a conceptual level —

that applying SDP to the 0-stages-to-go value function (i.e., $vCase^0 = rCase$, given previously) yields the following 1- and 2-stages-to-go value functions in the BoxWorld domain ($\neg$“ indicating the conjunction of the negation of all higher value partitions):

$$vCase^1 = \begin{array}{|lr|}\hline \exists b.BoxIn(b, paris) & : 19.0 \\ \hline \neg\text{“} \wedge \exists b, t.TruckIn(t, paris) \wedge BoxOn(b, t) & : \ 9.0 \\ \hline \neg\text{“} & : \ 0.0 \\ \hline \end{array}$$

$$vCase^2 = \begin{array}{|lr|}\hline \exists b.BoxIn(b, paris) & : 27.1 \\ \hline \neg\text{“} \wedge \exists b, t.TruckIn(t, paris) \wedge BoxOn(b, t) & : 17.1 \\ \hline \neg\text{“} \wedge \exists b, c, t.BoxOn(b, t) \wedge TruckIn(t, c) & : \ 8.1 \\ \hline \neg\text{“} & : \ 0.0 \\ \hline \end{array}$$

After sufficient iterations of SDP, the t-stages-to-go value function converges, giving the optimal value function (and corresponding policy) from Fig. 2.

Applications

(Variants of) symbolic dynamic programming have been successfully applied in decision-theoretic planning domains such as BlocksWorld, BoxWorld, ZenoWorld, Elevators, Drive, PitchCatch, and Schedule. The FOALP system [6] was runner-up at the probabilistic track of the 5th International Planning Competition (IPC-6). Related techniques have been used to solve path planning problems within robotics and instances of real-time strategy games, Tetris, and Digger.

Future Directions

The original *symbolic dynamic programming (SDP)* [1] approach is a value iteration algorithm for solving FOMDPs based on Reiter's situations calculus. Since then, a variety of exact algorithms have been introduced to solve MDPs with relational (RMDP) and first-order (FOMDP) structure.[3] *First-order value iteration (FOVIA)* [7, 8] and the *relational Bellman algorithm*

[3] We use the term *relational MDP* to refer to models that allow implicit existential quantification, and *first-order MDP* for those with explicit existential and universal quantification.

(ReBel) [9] are value iteration algorithms for solving RMDPs. In addition, *first-order decision diagrams (FODDs)* have been introduced to compactly represent case statements and to permit efficient application of symbolic dynamic programming operations to solve RMDPs via value iteration [10] and policy iteration [11]. All of these algorithms have some form of guarantee on convergence to the (ϵ-)optimal value function or policy. The expressiveness of FOMDPs has been extended to support indefinitely factored reward and transition functions in FOMDPs [12].

A class of linear-value approximation algorithms have been introduced to approximate the value function as a linear combination of weighted basis functions. *First-order approximate linear programming (FOALP)* [6] directly approximates the FOMDP value function using a linear program. *First-order approximate policy iteration (FOAPI)* [13] approximately solves for the FOMDP value function by iterating between policy and value updates in a policy-iteration style algorithm. Somewhat weak error bounds can be derived for a value function approximated via FOALP [6] while generally stronger bounds can be derived from the FOAPI solution [13].

Finally, there are a number of heuristic solutions to FOMDPs and RMDPs. *Approximate policy iteration* [14] induces rule-based policies from sampled experience in small-domain instantiations of RMDPs and generalizes these policies to larger domains. In a similar vein, *inductive policy selection using first-order regression* [15] uses the action regression operator in the situation calculus to provide the first-order hypothesis space for an inductive policy learning algorithm. *Approximate linear programming (for RMDPs)* [16] is an approximation technique using linear program optimization to find a best-fit value function over a number of sampled RMDP domain instantiations.

Recommended Readings

[1] Boutilier, C., Reiter, R., Price, B.: Symbolic dynamic programming for first-order MDPs. In: IJCAI-01, Seattle (2001) 690–697

[2] Fikes, R.E., Nilsson, N.J.: STRIPS: A new approach to the application of theorem proving to problem solving. AI Journal **2** (1971) 189–208

[3] Kushmerick, N., Hanks, S., Weld, D.: An algorithm for probabilistic planning. Artificial Intelligence **76** (1995) 239–286

[4] Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York (1994)

[5] Bellman, R.E.: Dynamic Programming. Princeton University Press, Princeton, NJ (1957)

[6] Sanner, S., Boutilier, C.: Approximate linear programming for first-order MDPs. In: UAI-2005, Edinburgh, Scotland (2005)

[7] Hölldobler, S., Skvortsova, O.: A logic-based approach to dynamic programming. In: In AAAI-04 Workshop on Learning and Planning in MDPs, Menlo Park, CA (2004) 31–36

[8] Karabaev, E., Skvortsova, O.: A heuristic search algorithm for solving first-order MDPs. In: UAI-2005, Edinburgh, Scotland (2005) 292–299

[9] Kersting, K., van Otterlo, M., de Raedt, L.: Bellman goes relational. In: ICML-04, Banff, Alberta, Canada, ACM Press (2004)

[10] Wang, C., Joshi, S., Khardon, R.: First order decision diagrams for relational MDPs. In: IJCAI, Hyderabad, India (2007)

[11] Wang, C., Khardon, R.: Policy iteration for relational MDPs. In: UAI, Vancouver, Canada (2007)

[12] Sanner, S., Boutilier, C.: Approximate solution techniques for factored first-order MDPs. In: ICAPS-07. (2007)

[13] Sanner, S., Boutilier, C.: Practical linear evaluation techniques for first-order MDPs. In: UAI-2006, Boston, Mass. (2006)

[14] Fern, A., Yoon, S., Givan, R.: Approximate policy iteration with a policy language bias. In: NIPS-2003, Vancouver (2003)

[15] Gretton, C., Thiebaux, S.: Exploiting first-order regression in inductive policy selection. In: UAI-04, Banff, Canada (2004) 217–225

[16] Guestrin, C., Koller, D., Gearhart, C., Kanodia, N.: Generalizing plans to new environments in relational MDPs. In: IJCAI-03, Acapulco, Mexico (2003)